Fundamentele programării
Limbajul C
Programare procedurală

Sorin Milutinovici
sorinmilu@gmail.com

1. Programarea înainte de computere

2. Procesorul

3. Limbaje de programare

4. Paradigme de programare

5. Algoritmi, scheme logice și pseudocod

6. Funcții, proceduri, subrutine

7. Limbajul de programare C

7.1 Variabile

7.2 Operatori

7.3 Instrucțiuni decizionale

7.4 Repetiții

7.5 Funcții

7.6 Siruri, matrici

7.7 Variabile, pointeri și referințe

7.8 Siruri de caractere

7.9 Fișiere

7.10 Recursivitate

7.11 Structuri

7.12 Union, Enum

7.13 Liste înlănțuite

8. Semnificația octeților

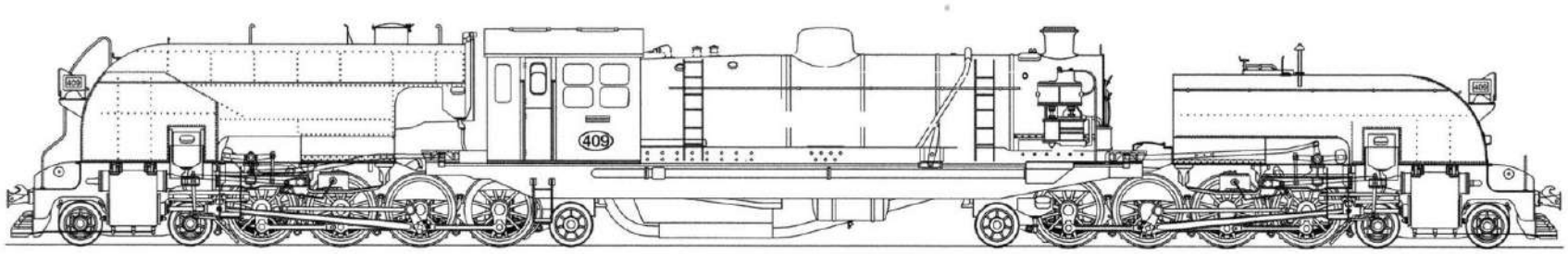
9. Încărcarea și execuția programelor în sistemele de operare

10. Extinderi ale paradigmei procedurale

11. Materiale suplimentare recomandate

Scopurile cursului

- O imagine coerentă asupra programării și a diferitelor moduri de abordare a acesteia
- Înțelegerea posibilităților și limitărilor programării și ale computerelor.
- Înțelegerea procesului de construcție a un program pe baza unei probleme definite.
- Învățarea principalelor elemente ale limbajului C



Resurse utile

Prezentările sunt disponibile în secțiunea “Files” pe teams

Cursurile vor fi înregistrate, înregistrările vor fi disponibile pe teams

Marea majoritate a exemplelor de cod sunt conectate cu editoare online de cod. Cel mai frecvent este OnlineGDB. Pentru cazul în care nu este disponibil, iată câteva alternative:

Toate fragmentele de cod din acest curs sunt disponibile pe github: https://github.com/sorinmilu/curs_programare

Comanda următoare necesită utilitarul “git” dar nu solicită nici o formă de autentificare, nici cont pe github.com.

```
git clone https://github.com/sorinmilu/curs_programare.git
```

Crearea unui user pe github.com (gratuit) permite consultarea online a fișierelor, fără a fi necesară descărcarea lor.

1. Programarea înainte de computere

Programarea este o consecință firească a eforturilor de a standardiza procesele de lucru pentru a le putea transmite și replica, efort absolut esențial pentru dezvoltarea socială. Gândirea algoritmică s-a dezvoltat cu mult înainte de apariția mașinilor care puteau executa liste de instrucțiuni iar acesta au apărut cu mult înaintea computerelor.

Rețetele culinare, descrierea modului în care se ascute o sabie, a modurilor de luptă, modul de preparare a berii sau a vopselelor pentru haine, toate au fost la un moment dat standardizate cu scopul de a putea fi replicate de altcineva sau transmise în timp.

De aceea, deși unii dintre noi pot fi mai atrași de sistemele standardizate de comunicare a instrucțiunilor decât alții, nu exista persoană care să fie "nepotrivită" pentru programare.

Naturalețea cu care ființa umană înțelege în cele din urmă programarea și relația intimă dintre mintea umană și programarea se observă și în cantitatea remarcabilă de jocuri și amuzamente care pot fi găsite în acest domeniu. Așa cum copiii, pe lângă învățarea limbajelor comune, își inventează limbaje pe care doar ei le vorbesc, programatorii crează limbaje "de jucărie" pe care le construiesc pentru a se amuza sau a demonstra faptul că este posibil. O parte din acestea se numesc limbaje ezoterice (sau pe scurt esolang).

Arc peste timp

Multe noțiuni de programare sunt prezentate prin analogie cu rețetele culinare. Richart Stallman, pionierul noțiunii de "software deschis" a folosit exact această metaforă pentru a explica de ce nu se poate patenta (în opinia sa) un program. Evident, există un limbaj de programare ezoteric care permite programarea în forma unei rețete culinare

1730BC

Cea mai veche rețetă culinară descoperită

- Se folosește carne.
- Pregătești apă.
- Adăugați sare fină, orz uscat, ceapă, eșalot persan și lapte.
- Zdrobești și adaugi praz și usturoi.

Colectia Babiloniana de la Universitatea Yale

CHEF este unul dintre limbajele de programare numite “ezoterice” (esolang) - limbaje de programare inventate pentru amuzament, nu cu scopul de a deveni cu adevărat instrumente de lucru

2002AC

Limbajul CHEF

Ingredients.
0 g fib1
1 g fib2
16 g iterator
16 g second iterator

```
IMethod.
Chop iterator.
Put fib2 into 1st mixing bowl.
Put fib2 into 1st mixing bowl.
Add fib1 into 1st mixing bowl.
Fold fib2 into 1st mixing bowl.
Fold fib1 into 1st mixing bowl.
Put fib1 into 1st mixing bowl.
Chop iterator until choped.
Mash second iterator.
Fold fib1 into 1st mixing bowl.
Put fib1 into 2nd mixing bowl.
Mash second iterator until mashed.
Pour contents of 2nd mixing bowl into the baking dish.
Serves 1
```

Limbajul de programare Chef

Un alt esolang: ARNOLD C

Un limbaj de programare în care instrucțiunile sunt reprezentate de fraze pe care Arnold Schwarzenegger le-a spus în diferite filme.
https://github.com/lhartikk/ArnoldC

```
IT'S SHOWTIME
HEY CHRISTMAS TREE avgMoviesPerYear
YOU SET US UP @I LIED
HEY CHRISTMAS TREE totalMovies
YOU SET US UP 56
HEY CHRISTMAS TREE years
YOU SET US UP 46
GET TO THE CHOPPER avgMoviesPerYear
HERE IS MY INVITATION avgMoviesPerYear
GET UP totalMovies
HE HAD TO SPLIT years
ENOUGH TALK
TALK TO THE HAND "Total Number of Movies done till date (2015 - via IMDB):"
TALK TO THE HAND totalMovies
TALK TO THE HAND "Avg Number of Movies/TV Shows Acted In Per Year, till date (2015 - via IMDB):"
TALK TO THE HAND avgMoviesPerYear
TALK TO THE HAND "Please note: The only variable type in ArnoldC is a 16bit integer. So the real average works out to be 1.2 movies per year"
YOU HAVE BEEN TERMINATED
```

Start program

Declară o variabilă de tip număr întreg

Atribuie o valoare

Print

Sfârșitul programului

Un program este o **suita de instrucțiuni** care sunt scrise de un **programator** pentru un **executant**

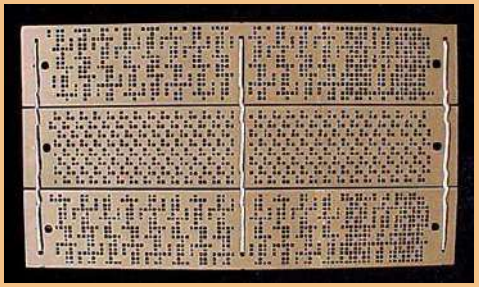
Ingrediente

- 1 1/2 căni maioneză
- 4 căței de usturoi, zdrobiți
- 1 linguriță sare
- 1 linguriță piper
- 2 lingurițe oregano
- 2 căni orez basmati
- 2 chili, feliați
- 1/3 cană apă
- 6 cartofi mai mici

Mod De Preparare

1. Amestecăm într-un caston maioneza cu sarea, piperul, usturoiul și oregano.
2. Puiul se acoperă cu toată această compoziție
3. Orezul se pune într-un vas de copt cu chili și se toarnă apoi apă peste.
4. Se adaugă cartofii și peste ei se pune puiul.
5. Se lasă la cuptor timp de 45 de minute la o temperatură de 180ºC.

1804 - Războiul de țesut Jacquard



https://www.youtube.com/watch?v=K6NgMNvK52A

00110101100101010100110011010
10010110011100101001100101 0
11001110100110001111001010 0
11100110010100110011001101 0
00110101100101010100110011010
11001110100110001111001010 0

2. Procesorul

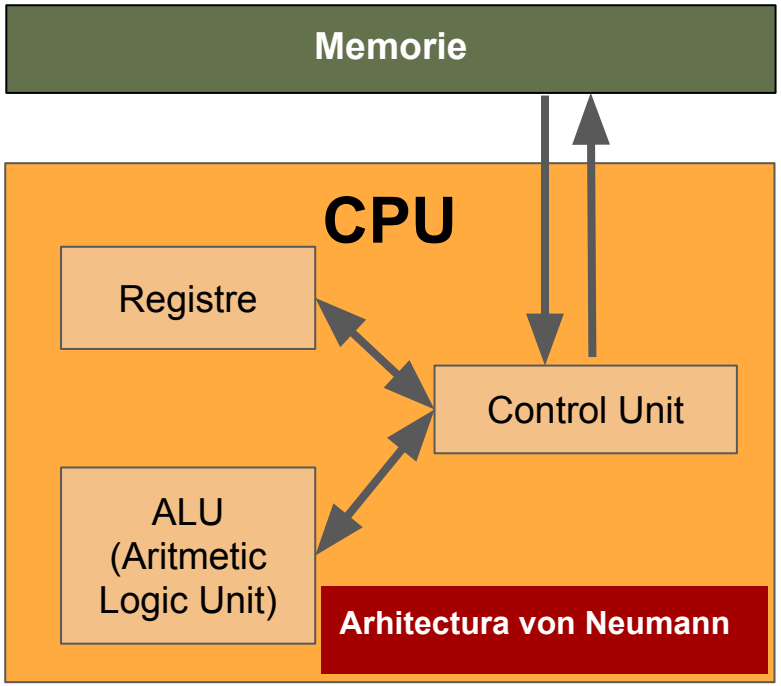
Procesorul reprezintă beneficiarul final (direct sau indirect) al programatorului. Există, evident, o serie întreagă de dispozitive mult mai interesante și mai amuzante decât procesorul dar orice program, chiar dacă este gândit pentru o imprimantă 3D, ajunge să se execute pe un procesor în relație cu memoria.

Execuția unei instrucțiuni implică citirea ei din memorie, decodificarea acesteia și execuția ei. Fiecare astfel de acțiune se face la semnalul ceasului sistemului. De aceea, cu cât ceasul sistemului este mai rapid (mai multe bătăi pe secundă) programul se va executa mai repede. Valorile numerice care însoțeau proesoarele până

Software = O listă de comenzi pentru procesor

Procesorul este în realitate beneficiarul nostru final, orice program, indiferent de modul în care este programat și de limbajul în care este programat va ajunge o simplă listă de comenzi pe care le execută procesorul

- Registre - memorii foarte rapide din care unitatea de control poate citi și în care poate scrie
- ALU - unitatea de calcule aritmetice
- Control Unit - unitatea care mută datele din memoria principală în regiștri și controlează ALU.



Un ciclu al procesorului mută datele pe traiectoria memorie -> registre -> ALU -> registre -> memorie.

Ordinele se numesc “instrucțiuni”

Program: R = 20 + 5

Programul este încărcat în memorie și procesat linie cu linie.

1. Copiază numărul stocat la adresa de memorie 10100 în registrul A (**LOAD**)
2. Adaugă numărul stocat la adresa de memorie 10101 celui stocat în registrul A (**ADD**)
3. Copiază conținutul registrului A la adresa de memorie 11111 (**STOR**)

Memorie

00000	10010100	LOAD 10100
00001	11010101	ADD 10101
00010	10111111	STOR 11111
00011	00000000	HALT
⋮		
10100	00010100	20
10101	00000101	5
10110	00000000	0
10111	00000000	0
11000	00000000	0

Procesorul va continua să execute toate liniile programului (instrucțiunile), una câte una, până când le termină.

Instruction Set Architecture

Totalitatea setului de instrucțiuni pe care le poate executa un anumit procesor se numește **Instruction Set Architecture**. Acest set de instrucțiuni depinde de tipul de procesor și **face parte din designul acestuia**. Setul este fix și este codificat hardware **în structura materială** a procesorului.

x86, MIPS, Itanium, Alpha, ARM, PowerPC, SPARC

https://en.wikipedia.org/wiki/Comparison_of_instruction_set_architectures

Probabil cel mai implementat set de instrucțiuni este x86, cu variantele lui pe 16, 32 și 64 de biți. Varietatea de implementări a determinat apariția unor grupuri de instrucțiuni suplimentare care nu fac parte din standardul de bază, numite extensii: **MMX** (extinderea registreor), **3DNow!** (calcul vectorial), **SSE** (extensii la operațiile cu numere în virgulă mobilă), **PAE** (extensii la managementul memoriei),

Programatorii, în special programatorii în limbajul C trebuie să aibă o imagine (oricât de generală) a modului in care funcționează procesorul. Implementările reale sunt extrem de complexe și constituie o specializare în sine. Pentru scopul acestui curs este suficient să folosim un model redus, un procesor virtual cu adrese pe 8 biti, cu un set de 8 instrucțiuni.

Un model extrem de simplificat (dar totuși funcțional) al unui procesor este implementat în programul numit SimHymn http://www.cburch.com/socs/hymn/index.html

Programul simulează un procesor cu dimensiunea cuvântului (cantitatea de date citită de-dată de procesot din memorie) 8 biți, cu o cantitate totală de memorie de 32 octeți, trei registre (Registrul de instrucțiuni, registrul acumulator și registrul de contor (Program Counter).

Procesorul are propriul limbaj de asamblare.

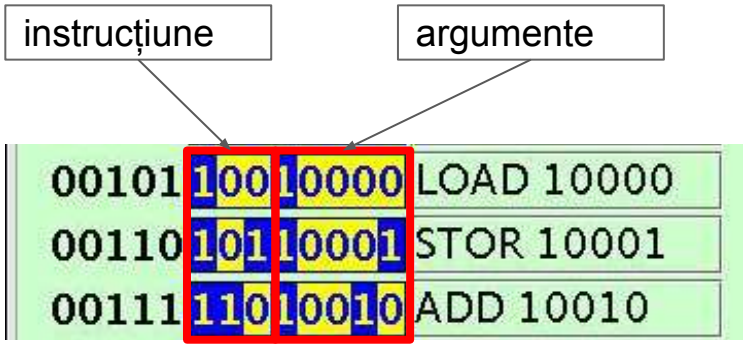
Zonele de memorie se pot încărca prin setarea cu mouse-ul al fiecărui bit sau prin încărcarea unui program scris în limbajul de asamblare al procesorului

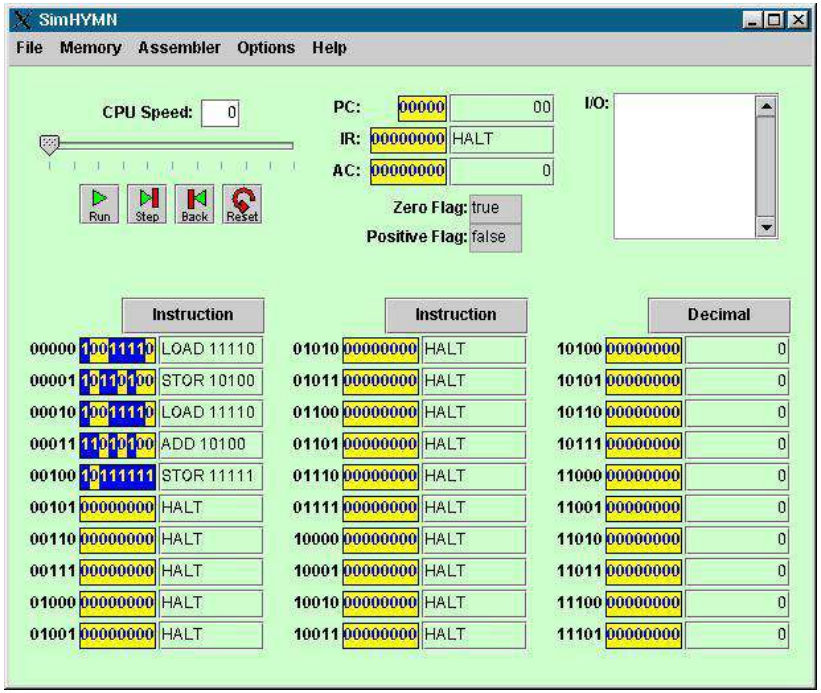
SimHYMN un program care simulează funcționarea unui procesor pe 8 biți

http://www.cburch.com/socs/hymn/index.html

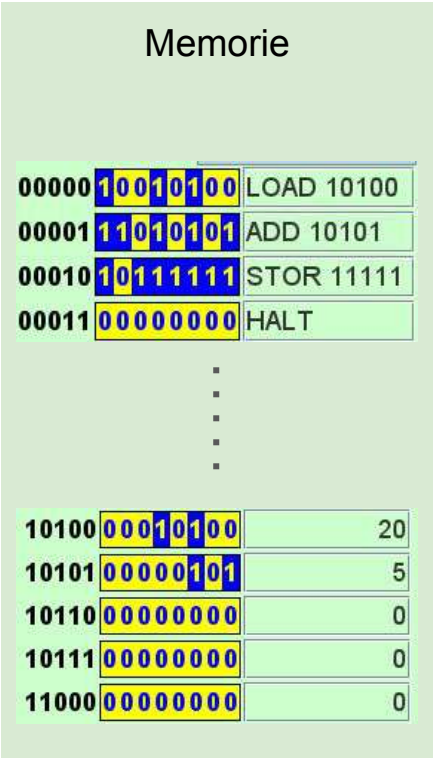
Numerele sunt stocate în 8 biți, cu semn

Intervalul posibil: -128 până la +127





Instrucțiuni



LOAD, ADD, STOR sunt instrucțiuni care fac parte din setul de instrucțiuni al procesorului. Sunt de obicei asociate unui pattern de biți: **LOAD**: 100, **ADD**: 110, **STOR**: 101

Programarea are ca scop final construirea unei liste de instrucțiuni în format binar însoțite de date, pe care procesorul le execută una câte una până ajunge la instrucțiunea **HALT** (sau una similară).

Scrierea directă a codului mașina este dificilă și nu este portabilă - codul nu poate rula decât pe procesorul pentru care a fost scris.

Limbajele de programare și sistemele de dezvoltare a aplicațiilor sunt instrumente proiectate pentru a ușura producerea codului mașină și a îl face mai portabil.

Toate limbajele de programare au elemente comune, acestea sunt toate determinate de felul în care procesorul execută codul mașină.

Setul de instrucțiuni SymHIMN

3 biti => 2^3 => 8 instructiuni

binar	nume	descriere
000	**HALT**	Oprește execuția programului
001	**JUMP**	Execută un salt la instrucțiunea de la adresa specificată
010	**JZER**	Jump on zero
011	**JPOS**	Jump on positive
100	**LOAD**	Copiază instrucțiunea de la adresa dată în acumulator
101	**STOR**	Copiază valoarea din acumulator în locația de memorie cu adresa dată
110	**ADD**	Adaugă valoarea din acumulator la valoarea stocată în adresa de memorie dată și copiază rezultatul în AC
111	**SUB**	Scade valoarea de la adresa de memorie dată din valoarea din acumulator

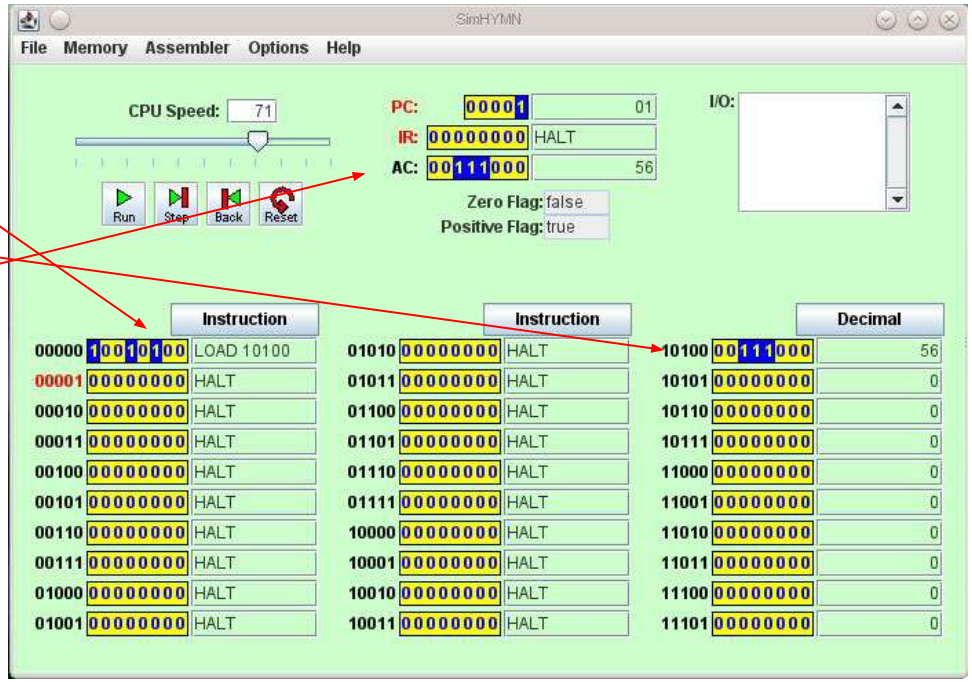
Adrese speciale de memorie		
	11110	Input de la consolă (Memory Mapped IO)
	11111	Output către consolă (Memory Mapped IO)

Program: Compararea a două numere

```
10011110
10101011
10011110
11101011
01001000
10001100
10111111
00000000
10001101
10111111
00000000
00000000
11111111
00000001
```

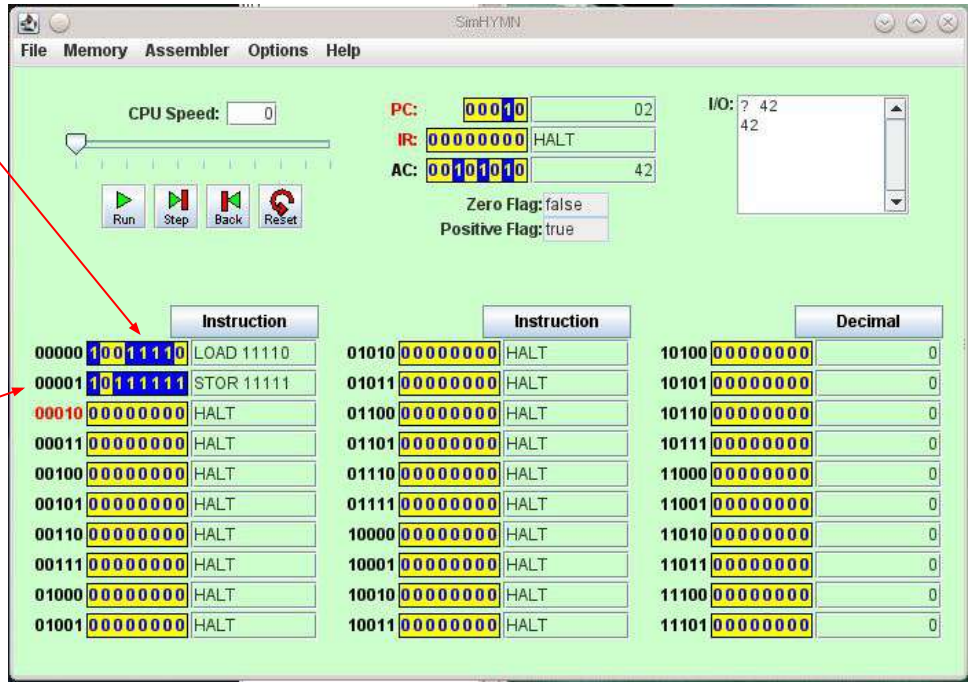
Instrucțiunea LOAD 10100
încarcă numărul stocat în adresa de memorie 10100
în acumulator

Instrucțiunea este văzută sub forma binară:
10010100



Adresa de memorie 11110
este o adresă specială, rezervată sistemului IO. Citirea acestei adrese determină așteptarea unei valori de la tastatură.

Adresa de memorie 11111
este o adresă specială, rezervată sistemului IO. Citirea acestei adrese determină afișarea valorii pe “display”



Mici programe în cod mașină

Adunare
10011110
10100110
10011110
11000110
10111111
00000000

Scădere
10011110
10100110
10011110
11100110
10111111
00000000

Compararea a două numere
10011110
10101011
10011110
11101011
01001000
10001100
10111111
00000000
10001101
10111111
00000000
00000000
11111111
00000001

Înmulțire
10010110
11110110
10110110
10011110
10110010
10011110
10110011
10110100
10010110
11010010
10110110
10010100
11110101
10110100
01101000
10010110
10111111
00000000
00000000
00000000
00000000
00000001
00000000

Împărțire
10010101
11110101
10110101
10011110
10110010
10011110
10110011
10010010
11110011
10110010
10110101
10010100
10110101
10010010
01110111
10010101
10111111
00000000
00000000
00000000
00000001
00000000

Limbajul de asamblare.

- Instrucțiunile binare (opcodes) sunt înlocuite cu instrucțiuni scurte text
- Se pot declara constante care vor fi încărcate în blocuri de memorie calculate automat
- Se pot declara blocuri de cod etichetate la care se pot face salturi.

Codul în limbajul de asamblare trebuie procesat de un program special (asamblor) care îl transformă în cod mașină.

Prima abstractizare a limbajului mașină - limbajul de asamblare

Adunare

adresă	valoare stocată la adresă	Instrucțiuni	Limbaj de asamblare	
00000	**100**11110	**LOAD**11110	READ	#Citește o valoare de la IO și o plasează în acumulator
00001	**101**00110	**STOR**00110	STOR Va	#Stochează valoarea citită în variabila Va
00010	**100**11110	**LOAD**11110	READ	#Citește o valoare de la IO și o plasează în acumulator
00011	**110**00110	**ADD** 00110	ADD Va	#Adaugă valoarea variabilei Va în acumulator
00100	**101**11111	**STOR**11111	WRITE	#Scrie valoarea din acumulator la IO
00101	**000**00000	**HALT**	HALT	#Opre□te programul
00110	00000000	00000000	Va:0	#Declară valoarea 0 pentru variabila Va

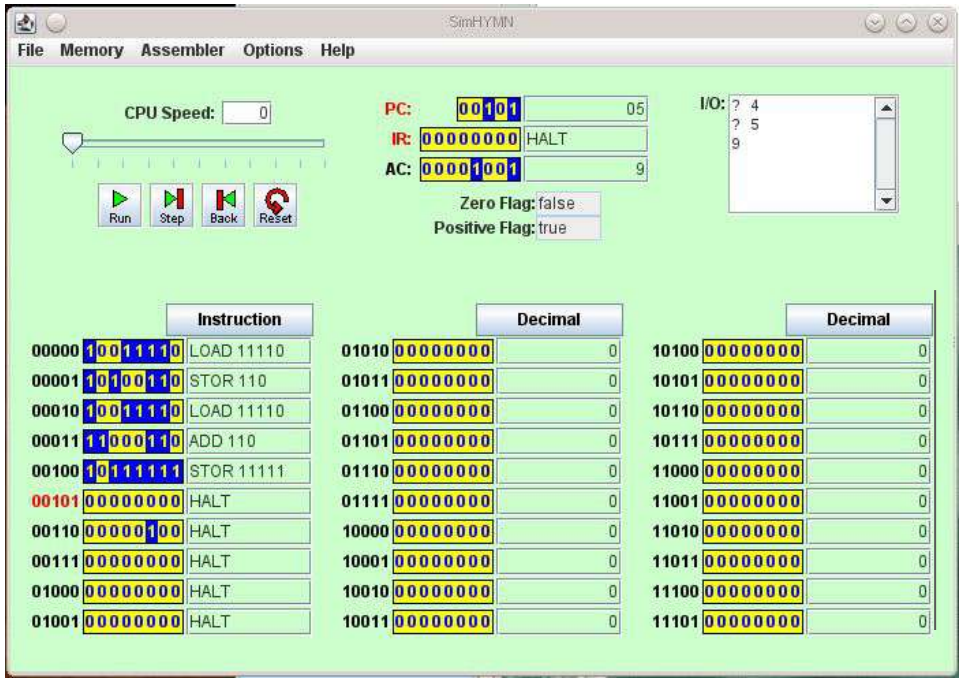
Scădere

adresă	valoare stocată la adresă	Instrucțiuni	Limbaj de asamblare	
00000	**100**11110	**LOAD**11110	READ	#Citește o valoare de la IO și o plasează în acumulator
00001	**101**00110	**STOR**00110	STOR Va	#Stochează valoarea citită în variabila Va
00010	**100**11110	**LOAD**11110	READ	#Citește o valoare de la IO și o plasează în acumulator
00011	**111**00110	**SUB** 00110	SUB Va	#Scade valoarea variabilei Va din acumulator
00100	**101**11111	**STOR**11111	WRITE	#Scrie valoarea din acumulator la IO
00101	**000**00000	**HALT**	HALT	#Opre□te programul
00110	00000000	00000000	Va:0	#Declară valoarea 0 pentru variabila Va

Limbajul de asamblare SymHimn

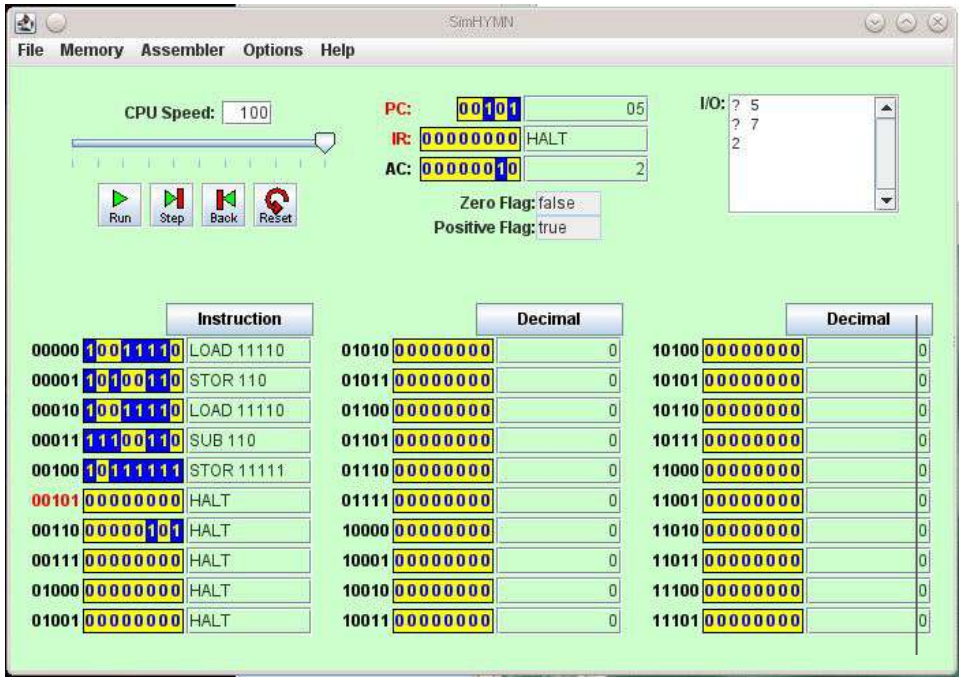
Adunare

adresă	valoare stocată la adresă	
00000	10011110	READ
00001	10100110	STOR Va
00010	10011110	READ
00011	11000110	ADD Va
00100	10111111	WRITE
00101	00000000	HALT
00110	00000000	Va:0



Scădere

adresă	valoare stocată la adresă	
00000	10011110	READ
00001	10100110	STOR Va
00010	10011110	READ
00011	11100110	SUB Va
00100	10111111	WRITE
00101	00000000	HALT
00110	00000000	Va:0



Compararea a două numere

adresa	valoare stocată la adresă	Instrucțiuni	Limbaj de asamblare	
00000	**100**11110	**LOAD**11110	READ	#Citește o valoare de la IO și o plasează în acumulator
00001	**101**01011	**STOR**00110	STOR Va	#Stochează valoarea citită în variabila Va
00010	**100**11110	**LOAD**11110	READ	#Citește o valoare de la IO și o plasează în acumulator
00011	**111**01011	**SUB** 00110	SUB Va	#Adaugă valoarea variabilei Va în acumulator
00100	**010**01000	**JZER**10000	JZER CEQ	#Sare la EQ dacă acumulatorul este zero
00101	**100**01100	**LOAD**01100	LOAD neq	#Dacă acumulatorul nu e zero, încarcă variabila neq în acumulator
00110	**101**11111	**STOR**11111	WRITE	#scrie valoarea din acumulator la IO
00111	00000000	**HALT**	HALT	#Opre□te programul
			CEQ:	
01000	**100**01101	**LOAD**01101	LOAD eq	#Încarcă eq în acumulator
01001	**101**11111	**STOR**11111	WRITE	#scrie valoarea din acumulator la IO
01010	**000**00000	**HALT**	HALT	#Opre□te programul
01011	00000000	00000000	Va:0	#declară valoarea Va ca fiind 0
01100	11111111	11111111	neq:-1	#declară valoarea neq ca fiind -1
01101	00000001	00000001	eq:1	#declară valoarea eq ca fiind 1

Programul tipărește -1 dacă două numere sunt inegale și +1 dacă cele două numere sunt egale

SimHYMN

File Memory Assembler Options Help

CPU Speed: 71

Run Step Back Reset

PC: 00101 05

IR: 00000000 HALT

AC: 11111111 -1

Zero Flag: false

Positive Flag: false

I/O: -1

	Instruction		Instruction		Decimal
00000	10010100 LOAD 10100	01010	10010111 LOAD 10111	10100	00010100 20
00001	11110101 SUB 10101	01011	10111111 STOR 11111	10101	00001100 12
00010	01001010 JZER 1010	01100	00000000 HALT	10110	11111111 -1
00011	10010110 LOAD 10110	01101	00000000 HALT	10111	00000001 1
00100	10111111 STOR 11111	01110	00000000 HALT	11000	00000000 0
00101	00000000 HALT	01111	00000000 HALT	11001	00000000 0
00110	00000000 HALT	10000	00000000 HALT	11010	00000000 0
00111	00000000 HALT	10001	00000000 HALT	11011	00000000 0
01000	00000000 HALT	10010	00000000 HALT	11100	00000000 0
01001	00000000 HALT	10011	00000000 HALT	11101	00000000 0

20 != 12

SimHYMN

File Memory Assembler Options Help

CPU Speed: 71

Run Step Back Reset

PC: 01100 0c

IR: 00000000 HALT

AC: 00000001 1

Zero Flag: false

Positive Flag: true

I/O: 1

	Instruction		Instruction		Decimal
00000	10010100 LOAD 10100	01010	10010111 LOAD 10111	10100	00010100 20
00001	11110101 SUB 10101	01011	10111111 STOR 11111	10101	00010100 20
00010	01001010 JZER 1010	01100	00000000 HALT	10110	11111111 -1
00011	10010110 LOAD 10110	01101	00000000 HALT	10111	00000001 1
00100	10111111 STOR 11111	01110	00000000 HALT	11000	00000000 0
00101	00000000 HALT	01111	00000000 HALT	11001	00000000 0
00110	00000000 HALT	10000	00000000 HALT	11010	00000000 0
00111	00000000 HALT	10001	00000000 HALT	11011	00000000 0
01000	00000000 HALT	10010	00000000 HALT	11100	00000000 0
01001	00000000 HALT	10011	00000000 HALT	11101	00000000 0

20 == 20

Înmulțire

adresa	valoare stocata la adresa	Instruc-țiuni	Limbaj de asamblare	
00000	**100**10110	**LOAD**10110	LOAD prod	#Reseteaza valoarea variabilei prod
00001	**111**10110	**SUB** 10110	SUB prod	#la 0 pentru a permite mai multe
00010	**101**10110	**STOR**10110	STOR prod	#rulari succesive ale programului
00011	**100**11110	**LOAD**11110	READ	#Citeste o valoare de la IO
00100	**101**10010	**STOR**10010	STOR Va	#Stocheaza valoarea citita in variabila Va
00101	**100**11110	**LOAD**11110	READ	#Citeste o valoare de la IO
00110	**101**10011	**STOR**10011	STOR Vb	#Stocheaza valoarea citita in variabila Vb
00111	**101**10100	**STOR**10100	STOR decr	#Stocheaza valoarea citita in variabila de decrementare
			LOOP:	#Porneste o bucla
01000	**100**10110	**LOAD**10110	LOAD prod	#Incarca variabila prod in acumulator
01001	**110**10010	**ADD** 10010	ADD Va	#Adauga variabila Va
01010	**101**10110	**STOR**10110	STOR prod	#Stocheaza noua valoare a prod
01011	**100**10100	**LOAD**10110	LOAD decr	#Incarca variabila de decrementare in acumulator
01100	**111**10101	**SUB** 10101	SUB one	#Scade valoarea variabilei one (1) din acumulator
01101	**101**10100	**STOR**10100	STOR decr	#Copiaza in memorie descr
01110	**011**01000	**JPOS**01000	JPOS LOOP	#Sare inapoi la LOOP daca val din acumulator e pozitiva
01111	**100**10110	**LOAD**10110	LOAD prod	#Incarca Va in acumulator
10000	**101**11111	**STOR**11111	WRITE	#Scrie valoarea din acumulator la IO
10001	**000**00000	**HALT**	HALT	#Opreste programul
10010	00000000	00000000	Va:0	#declara valoarea 0 pentru variabila Va
10011	00000000	00000000	Vb:0	#declara valoarea 0 pentru variabila Vb
10100	00000000	00000000	decr:0	#declara valoarea 0 pentru variabila decr
10101	00000001	00000001	one:1	#declara valoarea 1 pentru variabila one
10110	00000000	00000000	prod:0	#declara valoarea 0 pentru variabila prod

SimHYMN

File Memory Assembler Options Help

CPU Speed: 100

Run Step Back Reset

PC:	00000	00
IR:	00000000	HALT
AC:	00000000	0

Zero Flag: true

Positive Flag: false

I/O:

	Instruction	
00000	10010110	LOAD 10110
00001	11110110	SUB 10110
00010	10110110	STOR 10110
00011	10011110	LOAD 11110
00100	10110010	STOR 10010
00101	10011110	LOAD 11110
00110	10110011	STOR 10011
00111	10110100	STOR 10100
01000	10010110	LOAD 10110
01001	11010010	ADD 10010

	Instruction	
01010	10110110	STOR 10110
01011	10010100	LOAD 10100
01100	11110101	SUB 10101
01101	10110100	STOR 10100
01110	01101000	JPOS 1000
01111	10010110	LOAD 10110
10000	10111111	STOR 11111
10001	00000000	HALT
10010	00000000	HALT
10011	00000000	HALT

		Decimal
10100	00000000	0
10101	00000001	1
10110	00000000	0
10111	00000000	0
11000	00000000	0
11001	00000000	0
11010	00000000	0
11011	00000000	0
11100	00000000	0
11101	00000000	0

Împărțire

adresă	valoare stocată la adresă	Instrucțiuni	Limbaj de asamblare
00000	10010101	LOAD10101	LOAD div #Reseteaza valoarea div la 0
00001	11110101	SUB 10101	SUB div #Pentru a permite rulari succesive
00010	10110101	STOR10101	STOR div ale programului
00011	10011110	LOAD11110	READ #Citeste o valoare de la IO
00100	10110010	STOR10010	STOR Va #Stocheaza valoarea citita in variabila Va
00101	10011110	LOAD11110	READ #Citeste o valoare de la IO
00110	10110011	STOR10011	STOR Vb #Stocheaza valoarea citita in variabila Vb
			LOOP: #Porneste o bucla
00111	10010010	LOAD10010	LOAD Va #Incarca variabila Va in acumulator
01000	11110011	SUB 10011	SUB Vb #Scade variabila Vb din valoarea din acumulator
01001	10110010	STOR10010	STOR Va #Copiaza in memorie Va
01010	10110101	LOAD10101	LOAD div #Incarca variabila div in acumulator
01011	10010100	ADD 10100	ADD one #Scade valoarea variabilei one (1) din acumulator
01100	10110101	STRO10101	STOR div #Copiaza in memorie div
01101	10010010	LOAD10010	LOAD Va #Incarca Va in acumulator pentru test
01110	01110111	JPOS10111	JPOS LOOP #Sare inapoi la LOOP daca val din acumulator e pozitiva
01111	10010101	LOAD10101	LOAD div #Incarca Va in acumulator
10000	10111111	STOR11111	WRITE #Scrie valoarea din acumulator la IO
10001	00000000	HALT	HALT #Opreste programul
10010	00000000	00000000	Va:0 #declara valoarea 0 pentru variabila Va
10011	00000000	00000000	Vb:0 #declara valoarea 0 pentru variabila Vb
10100	00000001	00000001	one:1 #declara valoarea 1 pentru variabila one
10101	00000000	00000000	div:0 #declara valoarea 0 pentru variabila prod

SimHYMN

File Memory Assembler Options Help

CPU Speed: 100

Run Step Back Reset

PC:	00000	00
IR:	00000000	HALT
AC:	00000000	0

Zero Flag: true

Positive Flag: false

I/O:

		Instruction
00000	10010101	LOAD 10101
00001	11110101	SUB 10101
00010	10110101	STOR 10101
00011	10011110	LOAD 11110
00100	10110010	STOR 10010
00101	10011110	LOAD 11110
00110	10110011	STOR 10011
00111	10010010	LOAD 10010
01000	11110011	SUB 10011
01001	10110010	STOR 10010

		Instruction
01010	10010101	LOAD 10101
01011	11010100	ADD 10100
01100	10110101	STOR 10101
01101	10010010	LOAD 10010
01110	01100111	JPOS 111
01111	10010101	LOAD 10101
10000	10111111	STOR 11111
10001	00000000	HALT
10010	00000000	HALT
10011	00000000	HALT

		Decimal
10100	00000001	1
10101	00000000	0
10110	00000000	0
10111	00000000	0
11000	00000000	0
11001	00000000	0
11010	00000000	0
11011	00000000	0
11100	00000000	0
11101	00000000	0

Adăugarea a două numere în x86 TASM

```
Data segment

msg db 0dh,0ah,"Introduceti primul numar: $"
msg1 db 0dh,0ah,"Introduceti al doilea numar: $"
suma db 0dh,0ah,"Rezultatul este: $"

Data ends
Code segment
assume CS:Code,DS:Data
start:
mov ax,Data ; muta Data in registrul ax
mov DS,ax
mov dx,offset msg ; Afiseaza msg
mov ah,09h  ; muta 09h in registrul de intreruperi
int 21h ; apeleaza codul de intrerupere
mov ah,01h ; muta 01h in registrul de intreruperi
int 21h  ; apeleaza codul de intrerupere
sub al,30h ; transforma din ascii in numar
mov bl,al

mov dx,offset msg1 ; Afiseaza msg1
mov ah,09h
int 21h
mov ah,01h ; codul de intrerupere pentru input
int 21h
sub al,30h ; transforma din ascii in numar
mov cl,al

add bl,cl ; Adauga cele doua numere
mov dx,offset suma ; afiseaza stringul "suma"
mov ah,09h
int 21h
mov cl,bl ; stocheaza valoarea sumei in cl
call AsciiConv ; converteste la ASCII pentru afisare
mov dl,bl ; muta in registrul stream
mov ah,02h  ; seteaza comanda de intrerupere
int 21h ; lanseaza intreruperea
mov ah,4ch ; seteaza comanda de intrerupere de final
int 21h ; lanseaza intreruperea

AsciiConv proc ; < 0a adaugam 30h
 cmp bl,0ah ; >=  0a adaugam inca  07h
 jc skip
 add bl,07h
 skip: add bl,30h
 ret
 endp

Code ends
end start
```

fragment din tabela ASCII

2E	.
2F	/
30	0
31	1
32	2
33	3
34	4
35	5
36	6
37	7
38	8
39	9
3A	:
3B	;

Scăzând 30h din valuarea ascii a unui număr obținem numărul intreg corespunzător

Cateva coduri ale funcțiilor int 21h (restul aici)	
09h	Scrie un sir la consola
01h	Citeste un caracter de la consola
02h	Scrie un caracter la consola
4ch	Iese din program

Registre generale

`ax` – **accumulator register**, acumulator, 16 bit
`bx` – **base register**. Utilizat ca pointer catre datele localizate in registrul de segmente DS, cand se lucreaza in modul segmentat
`cx` – **counter register** - registrul de numarare. Utilizat in operatii de shift/rotate si bucle
`dx` – **data register,** utilizat in operatii aritmetice si de input/output
`si` – **source index register.** Utilizat ca pointer catre o sursa in operatii de tip stream
`di` – **destination index**. Pointer catre destinatie in operatii de tip stream
`bp` – **stack base pointer** - trimite catre baza stivei
`sp` – **stack pointer** - trimite catre topul stivei

Registre ale segmentelor

`ds` – **data segment**, pointer catre segmentul de date
`es` – **extra segment,** pointer catre o zona suplimentara de date
`ss` – **stack segment**, pointer catre stiva
`cs` – **code segment** - pointer catre cod

ip - instruction pointer

Flags

`c` – **Carry Flag**. Set if the last arithmetic operation carried (addition) or borrowed (subtraction) a bit beyond the size of the register. This is then checked when the operation is followed with an add-with-carry or subtract-with-borrow to deal with values too large for just one register to contain.
`z` – **Zero Flag**. Set if the result of an operation is Zero (0).
`s` – **Sign Flag**. Set if the result of an operation is negative.
`o` – **Overflow Flag**. Set if signed arithmetic operations result in a value too large for the register to contain.
`p` – **Parity Flag**. Set if the number of set bits in the least significant byte is a multiple of 2.
`a` – **Adjust Flag**. Carry of Binary Code Decimal (BCD) numbers arithmetic operations.
`i` – **Interruption Flag**. Set if interrupts are enabled.
`d` – **Direction Flag**. Stream direction. If set, string operations will decrement their pointer rather than incrementing it, reading memory backwards.

http://ref.x86asm.net/

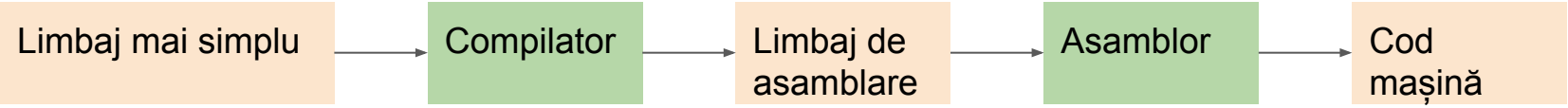
Dezavantajele programării în cod mașină

- Codul binar este foarte dificil de citit
- Comenzile disponibile sunt puține și primitive
- Programele sunt foarte lungi și foarte greu de depanat
- Limbajul mașină nu oferă funcții matematice, ele trebuie scrise de fiecare dată
- Locațiile de memorie trebuie manipulate direct făcând imposibil lucrul cu cantități mari de memorie

Dezavantajele programării în limbaj de asamblare

- Limbajul de asamblare este în continuare foarte dificil de citit
- Comenzile disponibile nu sunt mult mai multe decât cele în cod mașină
- Programele sunt foarte lungi și foarte greu de depanat
- Programele scrise în limbaj de asamblare sunt dependente de sistemul de operare și de arhitectură (asamblorul este un program ca oricare altul și rulează doar pe un anumit sistem de operare.

Se poate construi un limbaj mai simplu împreună cu un “traducător” care să genereze automat limbaj de asamblare?



3. Limbaje de programare

Cum ar trebui să arate?

1. Variabilele trebuiesc inițializate
2. Compilatorul trebuie să știe ce vom stoca în ele (numere întregi, șiruri)

```
altx = 10;
```

```
int altx = 10;
```

3. Putem concepe un sistem de teste la care programul să sară la diverse adrese de memorie, executând diferite secvențe de cod.

```
if( altx < 20 ) {
   /* executa bloc a */
 } else {
   /* executa bloc b */
 }
```

4. Putem concepe un sistem de instrucțiuni care să permită execuția repetată a unei anumite secvențe de instrucțiuni

```
for( a = 10; a < 20; a = a + 1 ){
    printf("value of a: %d\n", a);
 }
```

5. Putem concepe un sistem prin care să înregistrăm secvențe de instrucțiuni la anumite adrese de memorie, dincolo de instrucțiunea HALT, care să fie executate dacă sunt apelate din interiorul codului principal

```
int max(int num1, int num2) {

   /* local variable declaration */
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
   return result;
}
```

Limbaje de nivel înalt

- Limbaj mult mai apropiat de limbajul natural
- Operații matematice predefinite care pot fi apelate prin operatori simpli (>, <, *, /, ^ etc.)
- Operații logice predefinite (AND, OR etc.)
- Instrucțiuni care definesc blocuri de cod repetitive, condiționale
- Suport automat pentru operații de Input/Output comune.
- Diferite posibilități adăugate limbajului în funcție de scopul pentru care a fost creat.

Ethnologue - languages of the world (https://www.ethnologue.com/guides/how-many-languages) înregistrază 7,139 limbi vorbite pe pământ.

Online Historical Encyclopaedia of Programming Languages (http://hopl.info/) înregistrază 8945 limbaje de programare.

De ce sunt atât de multe limbaje de programare?

Absolut toate ajung aici:

```
10011110
10101011
10011110
11101011
01001000
10001100
10111111
00000000
10001101
10111111
00000000
00000000
11111111
00000001
```

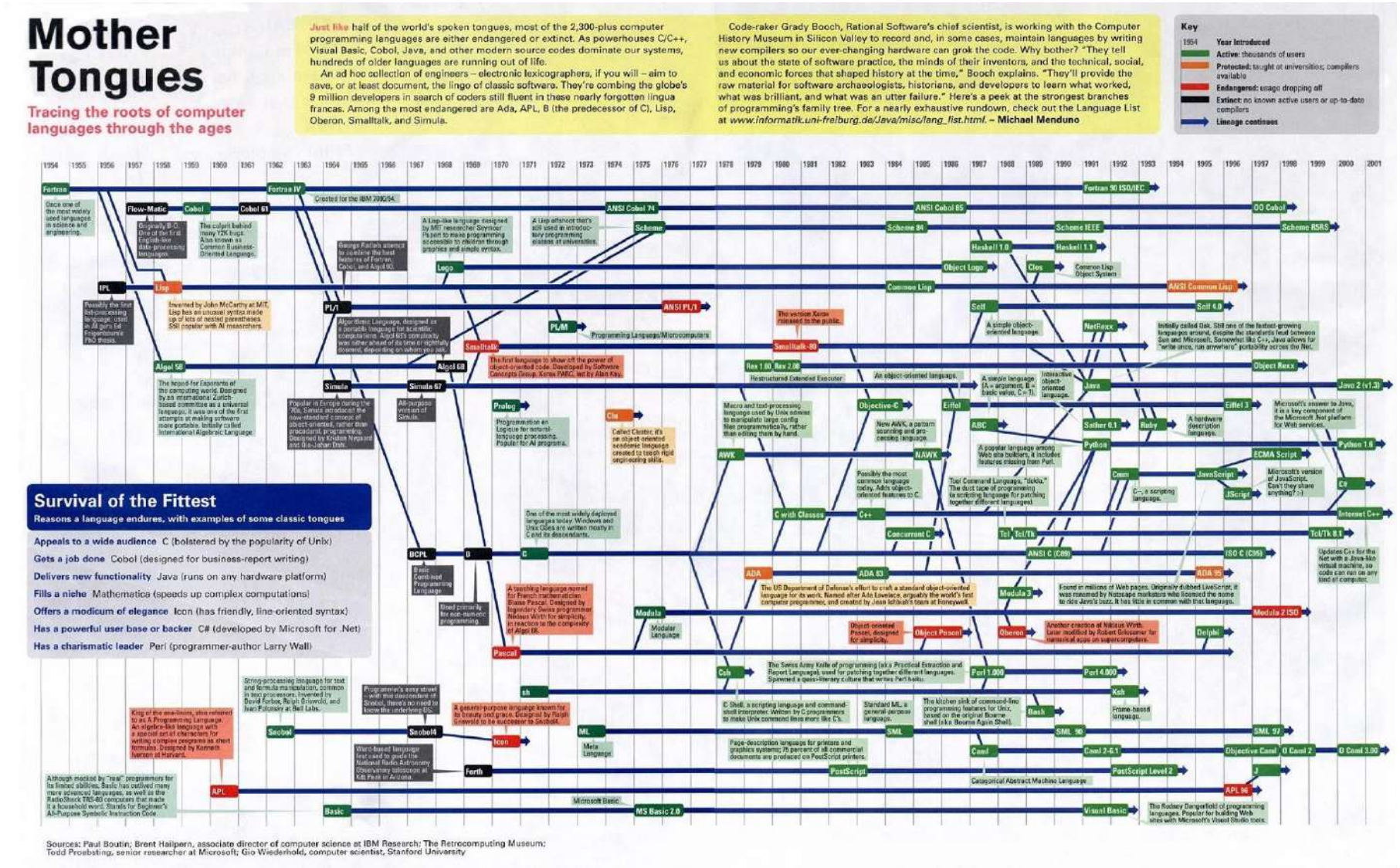
Fiecare program exprima una sau mai multe idei.

Crearea unui program nu are ca scop doar automatizarea unui procesor ci, de multe ori mult mai important, exprimarea unor idei și concepte.

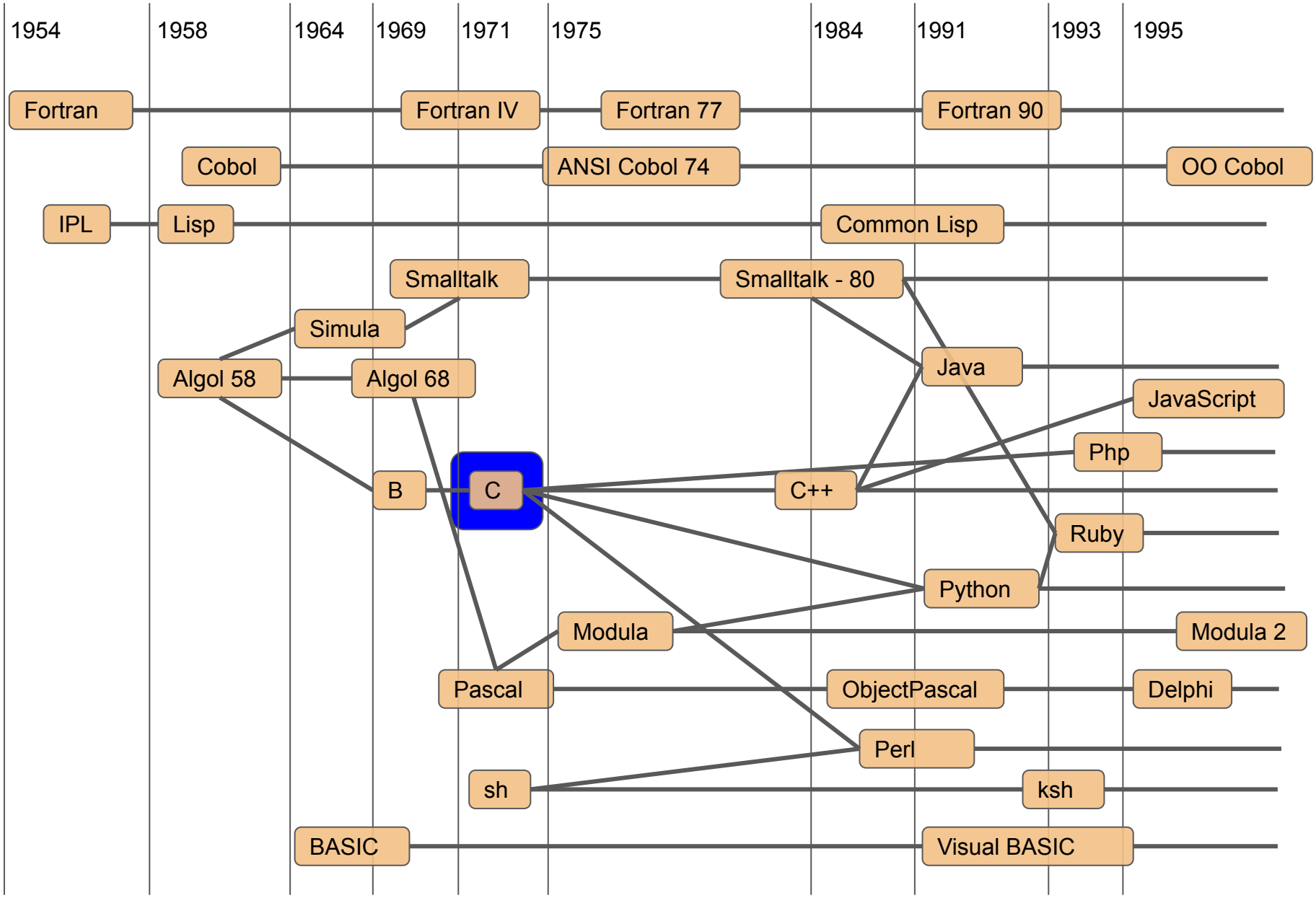
Limbajul este limitat iar exprimarea conceptelor este dependentă de el. Obiceiul limbii germane de a înlănțui cuvinte permite exprimarea eficientă unor concepte complicate (cum este celebrul Schadenfreude care este compus din Schaden (stricăciune, defecțiune) și freude (bucurie). Alte limbi au dificultăți în exprimarea aceluiași concept.

Toate limbajele de programare, toate paradigmele în care acestea se găsesc, toate sintaxele inventate în acest domeniu, toate acestea sunt pentru oameni, nu pentru computere. Ca atare, nu e neobișnuit să găsim în ele o componentă psihologică și socială semnificativă.

Evoluția limbajelor de programare



Evoluția limbajelor de programare



4. Paradigme de programare

Paradigmele sunt modele de rezolvare a unor probleme. Având în vedere că dezvoltarea unui program nu înseamnă doar generarea unei succesiuni de 1 și 0, ci exprimarea unor idei umane, e oarecum firesc ca de-a lungul timpului să se fi dezvoltat mai multe tipuri distincte de abordări pentru rezolvarea aceleiași probleme

Aceeași problemă se poate rezolva în toate stilurile posibile. Anumite paradigme sunt mai potrivite pentru anumite clase de probleme, altele pentru alte clase de probleme. În acest capitol vom urmări un exemplu dezvoltat prin toate paradigmele de programare prezentate.

Conceptul a fost definit de Robert Floyd și a fost prezentat într-o comunicare ținută cu ocazia primirii premiului Turing. Comunicarea poate fi citită aici

Paradigme de programare

imperative

Programele sunt compuse din instrucțiuni care spun computerului ce are de făcut și prin ce metode să ajungă la scopurile definite. Operația principală este atribuirea unor valori variabilelor definite.

- procedurale: Programele sunt împărțite în mai multe proceduri care se pot apela una pe cealaltă.
- Orientate pe obiecte: Programele sunt definite ca o sumă de componente care transmit mesaje între ele. Componentele pot fi extinse cu noi caracteristici dând naștere unor noi obiecte în timpul rulării.
- Orientate pe eveniment: Programul este o lista a tuturor evenimentelor posibile, cu acțiunile corespunzătoare fiecăruia.

declarative

Programele sunt compuse din instrucțiuni care spun computerului ce are de făcut fără să menționeze modul.

- DSL (domain specific language): Colectii de funcții special proiectate pentru un anumit domeniu de interes cu utilizare restrânsă
- funcționale: Programele sunt definite ca o colecție de funcții matematice.
- logice: Programele sunt definite ca o succesiune de axiome urmate de relații logice între date. Utilizate în special în lucrul cu colecții mari de informație cu relații cunoscute.

Programare imperativă procedurală (C)

Apelarea procedurilor

```c
#include <stdio.h>

int putere(int m, int n);

int main()
{
   int a=3, b=5;
   int c = putere(a, b);
   printf("%d la puterea %d este %d\n",a,b,c);
   return 0;
}

int putere(int m, int n)
{
   int i, p=1;
   for (i=1; i<=n; i++) {
       p = p * m;
   }
   return p;
}
```

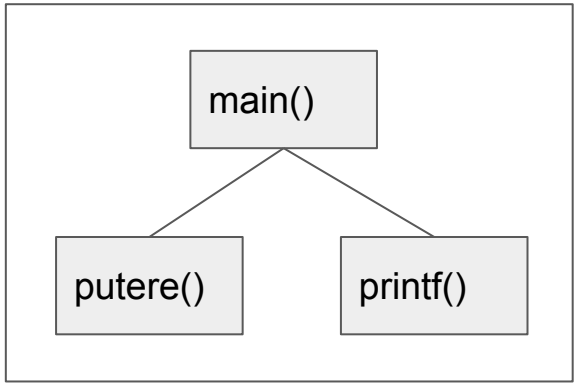
https://onlinegdb.com/ryyHCliLv

```c
int c = putere(a, b);
```

```c
printf("%d la puterea %d este %d\n",a,b,c);
```

putere_procedural.c

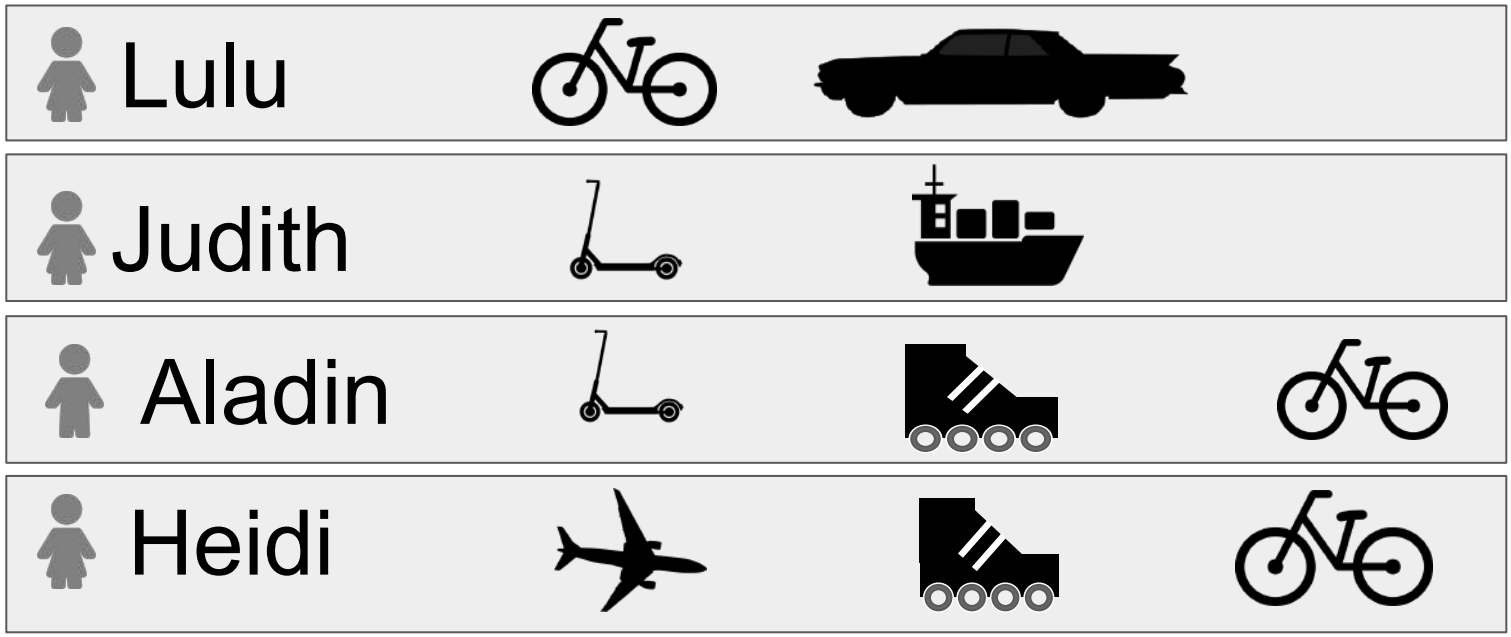
Proceduri

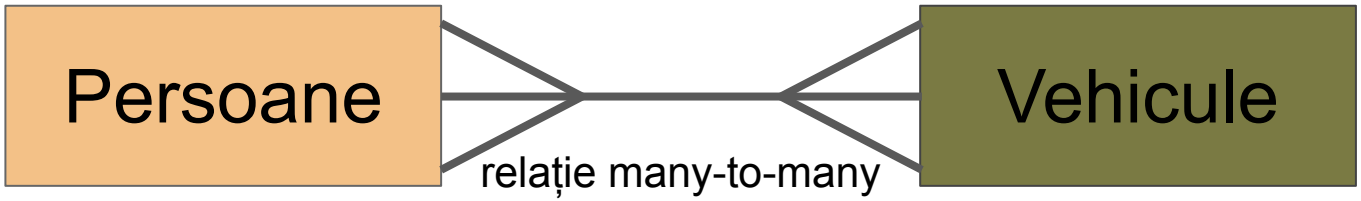



```
3 la puterea 5 este 243
```

O situație reală

Lulu conduce bicicletă și mașină, Judith conduce trotinetă și vapor, Aladin conduce trotinetă, role și bicicletă, Heidi conduce avion, role ai bicicletă





Cine conduce bicicleta?

Programare imperativă procedurală (javascript)

```
const conducatori = [
        { nume: "Lulu", conduce: ["bicicleta", "masina"] },
        { nume: "Judith", conduce: ["trotineta", "vapor"] },
        { nume: "Alladin", conduce: ["trotineta", "role", "bicicleta"] },
        { nume: "Heidi", conduce: ["avion", "role", "bicicleta"] }];
```

structura de date

```
var bicicleta = cineconduce("bicicleta");

print(bicicleta);

function cineconduce(vehicul) {
        let rezultate = [];
                for (i=0; i < conducatori.length; i++) {
                        for (j=0; j < conducatori[i].conduce.length; j++) {
                                if (conducatori[i].conduce[j] == vehicul) {
                                        rezultate.push(conducatori[i].nume);
                                }
                        }
                }
                return rezultate;
        }
}
```

https://onlinegdb.com/rJuzF3aLw

cod executabil format dintr-o funcție și un apel al acesteia din afara ei. Anumite limbaje de programare tratează orice parte a programului principal care nu se găsește într-o funcție ca făcând parte din funcția “main”

obs: **print** nu este o instrucțiune standard javascript, javascript nu are instrucțiuni de output, acestea depind de implementare, în acest caz Rhino

vehicule_procedural.js

Programare imperativă obiectuală

Programarea orientată obiect încearcă să modeleze programul ca pe o serie de entități (obiecte) care interacționează între ele prin mesaje. Conceptual, paradigma este foarte ofertanta pentru ca face programul sa semene cu un ecosistem natural, ecosistem pe care creierul uman e obișnuit să îl înțeleagă.

Un obiect se obține în timpul rulării programului pe baza clasei corespunzătoare (ceea ce se programează sunt clasele, nu obiectele)

Obiectul este o abstracțiune informatică, posibilă datorită modului de organizare a limbajului și a compilatorului, care încapsulează comportamentul și starea.

- **Starea** este specificată prin **atribute** (culoare, greutate etc.)
- **Comportamentul** este specificat prin **metode** (calculeaza_media, schimba_culoarea etc)

Obiectele comunică între ele prin **mesaje**

Construcția unei aplicații într-un mediu orientat obiect începe înainte de scrierea codului, cu o etapă serioasă de proiectare, etapă în care se stabilesc obiectele, atributele și metodele lor.

Într-o fermă orientată obiect există un fermier orientat obiect și o vacă orientată obiect. Fermierul mulge vaca. Laptele este:

?

- obiectul lapte
- mesaj între fermier și vacă

?

  - atașăm metoda “mulge” la obiectul fermier
  - atașăm metoda “da_lapte” la obiectul vacă

Programare imperativă obiectuală (C++)

```
#include <iostream>
using namespace std;

class Matematica
{
   float pi=3.14;

   public:
       int putere(int m, int n){
           int i, p=1;
           for (i=1; i<=n; i++) {
               p = p * m;
           }
           return p;
       }
};

int main()
{
  Matematica mate;
  int a=3;
  int b=5;
  int c = mate.putere(a,b);
  cout << a << " la puterea " << b << " este " << c;
}
```

Clasa - un prototip pe baza căruia se pot construi obiecte

Obiect - o instanță a clasei.

```
Variabila "mate" va contine un obiect de tip
"Matematica".

Matematica mate;

mate con□ine o metoda (putere) care se
apelează astfel:

int c = mate.putere(a,b);
```

putere_obiectual.cpp

https://onlinegdb.com/rJMrDfLww

Programare imperativă obiectuală - exemplu

```
int main()
{
    Vehicul bicicleta{ "Bicicleta" };
    Vehicul trotineta{ "Trotineta" };
    Vehicul masina{ "Masina" };
    Vehicul vapor{ "Vapor" };
    Vehicul role{ "Role" };
    Vehicul avion{ "Avion" };
    Conducator lulu{ "Lulu" };
    Conducator judith{ "Judith" };
    Conducator aladin{ "Alladin" };
    Conducator heidi{ "Heidi" };
    lulu.addVehicul(bicicleta);
    lulu.addVehicul(masina);
    judith.addVehicul(trotineta);
    judith.addVehicul(vapor);
    aladin.addVehicul(trotineta);
    aladin.addVehicul(role);
    aladin.addVehicul(bicicleta);
    heidi.addVehicul(avion);
    heidi.addVehicul(role);
    heidi.addVehicul(bicicleta);

    cout << bicicleta << '\n';
    return 0;
}
```

Instanțierea obiectelor de tip Vehicul

Instanțierea obiectelor de tip Conducător

Adăugarea vehiculelor la Conducători folosind metoda addVehicul.

cout imprima bicicleta la consolă

Putem “imprima” un obiect?
In principiu nu, dar se poate “supraîncărca” operatorul <<

vehicule_obiectual.cpp

https://onlinegdb.com/HyvTXorwP

Declararea unei clase (Conducator)

Atribute:

- nume (String m_name)
- un vector de Vehicule (m_Vehicul)

Metode:

- Conducator (constructor)
- addVehicul (functie)

```
class Conducator {
private:
   string m_name{};
   vector<reference_wrapper<const Vehicul>> m_Vehicul{};
public:
   Conducator(const string& name) :
       m_name{ name } { }
   void addVehicul(Vehicul& Vehicul);
   friend ostream& operator<<(ostream &out, const Conducator &Conducator);
   const string& getName() const { return m_name; }
};
```

```
void Conducator::addVehicul(Vehicul& Vehicul) {
   m_Vehicul.push_back(Vehicul);
   Vehicul.addConducator(*this);
}
```

```
ostream& operator<<(ostream &out, const Conducator &Conducator) {
   if (Conducator.m_Vehicul.empty()) {
       out << Conducator.m_name << " nu conduce nimic"; return out
   }
   out << Conducator.m_name << " conduce: ";
   for (const auto& Vehicul : Conducator.m_Vehicul)
       out << Vehicul.get().getName() << ' '; return out;
}
```

Supraîncărcarea operatorilor este importantă în programarea obiectuală - fiecare obiect are alt fel de structură. Egalitatea a două numere nu este la fel cu egalitatea a două vehicule.

Felul în care se imprimă un vehicul e diferit de felul în care se imprimă un aeroport.

Se poate evita prin scrierea metodelor specifice pentru fiecare clasa (ex: bicicleta.print())

Programare imperativă obiectuală

Structura claselor

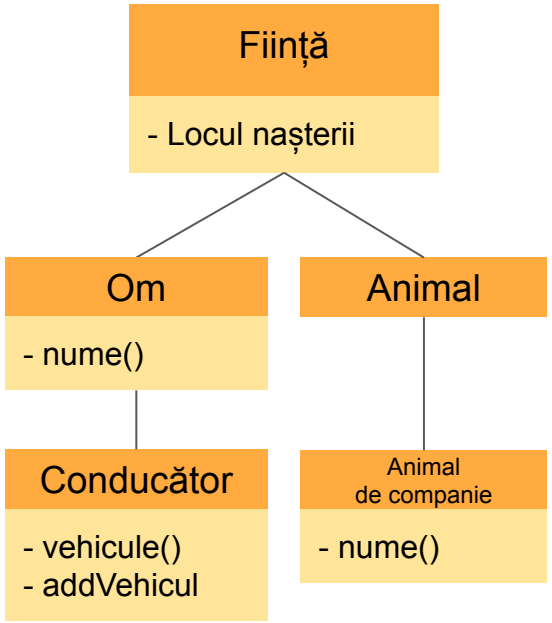
Conducator
- nume
- vehicule()
- addVehiucul

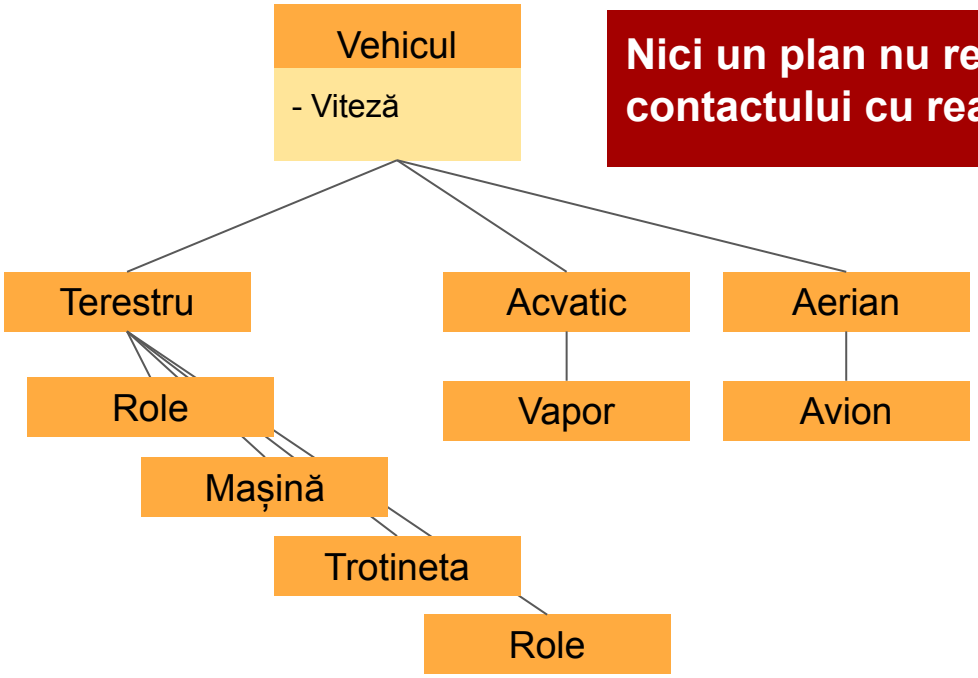
Vehicul
- nume
- vehicule()
- addConducator

Structura claselor este de obicei proiectată complet înainte de implementare. Ea se bazează pe abilitatea subclaselor de a moșteni proprietățile claselor părinte.

Este foarte tentantă încercarea de reprezentare cât mai corectă a realității în proiectarea unei structuri de clase. Aceasta duce de multe ori la o supraîncărcare a proiectului cu clase inutile.

Reproiectările sunt dificile dar foarte frecvente!!





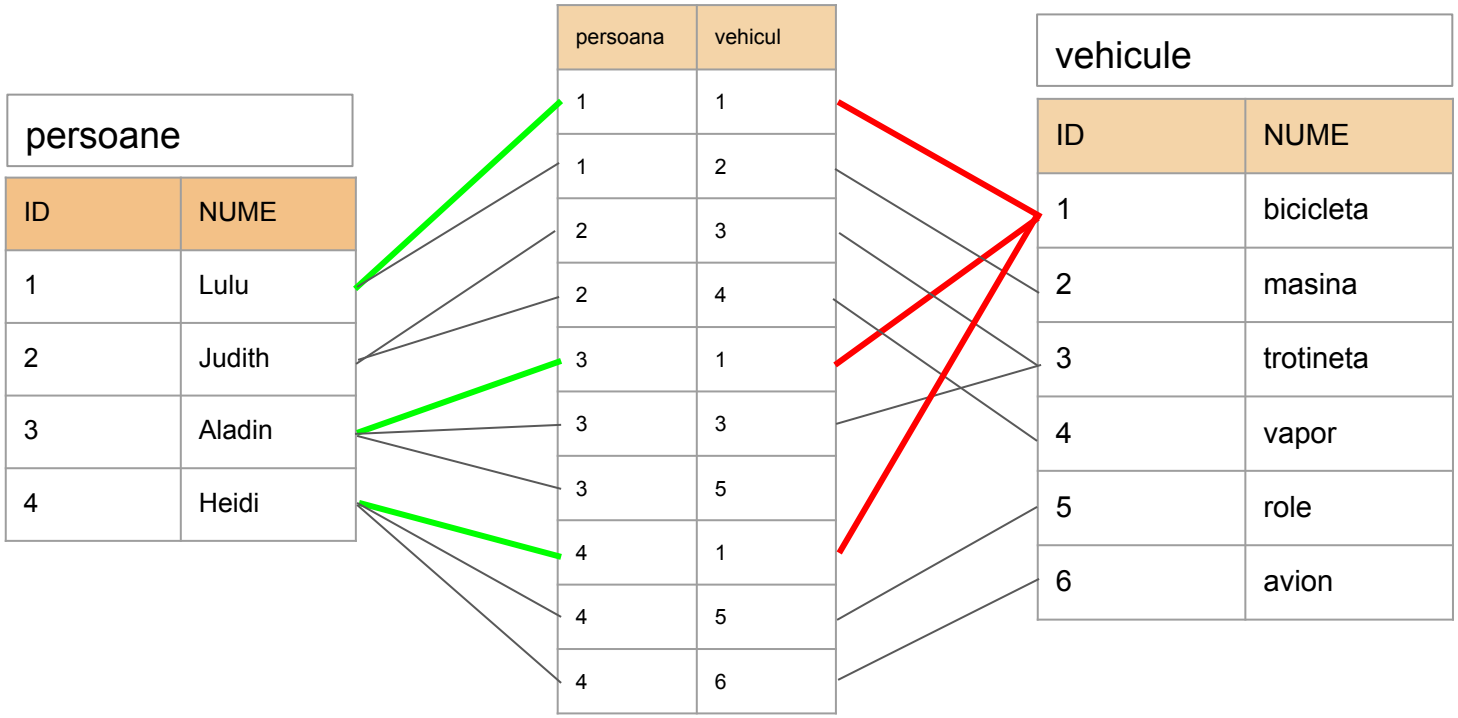
Nici un plan nu rezistă contactului cu realitatea

Toate subclasele obiectului “Ființă” vor moșteni atributul “locul nașterii”. Conducător va moșteni și nume (de la OM).

Toate subclasele obiectului “Vehicul” vor moșteni atributul “viteză”.

Programare declarativă cu DSL - SQL

SQL - Structured Query Language este un limbaj utilizat pentru extragerea datelor stocate in bazele de date relaționale. Acestea au o structură aparte - datele sunt stocate în tabele care sunt conectate între ele prin relații precis definite.



-

Lulu
Aladin
Heidi

conducatori.sql + conducatori_create_database.sql

Programare declarativa cu DSL - SQL

```
SELECT persoane.nume from persoane
JOIN persoane_vehicule on persoane_vehicule.persoana_id=persoane.id
JOIN vehicule on vehicule.id=persoane_vehicule.vehicul_id
WHERE vehicule.nume='bicicleta'
```

Echivalentul procedural

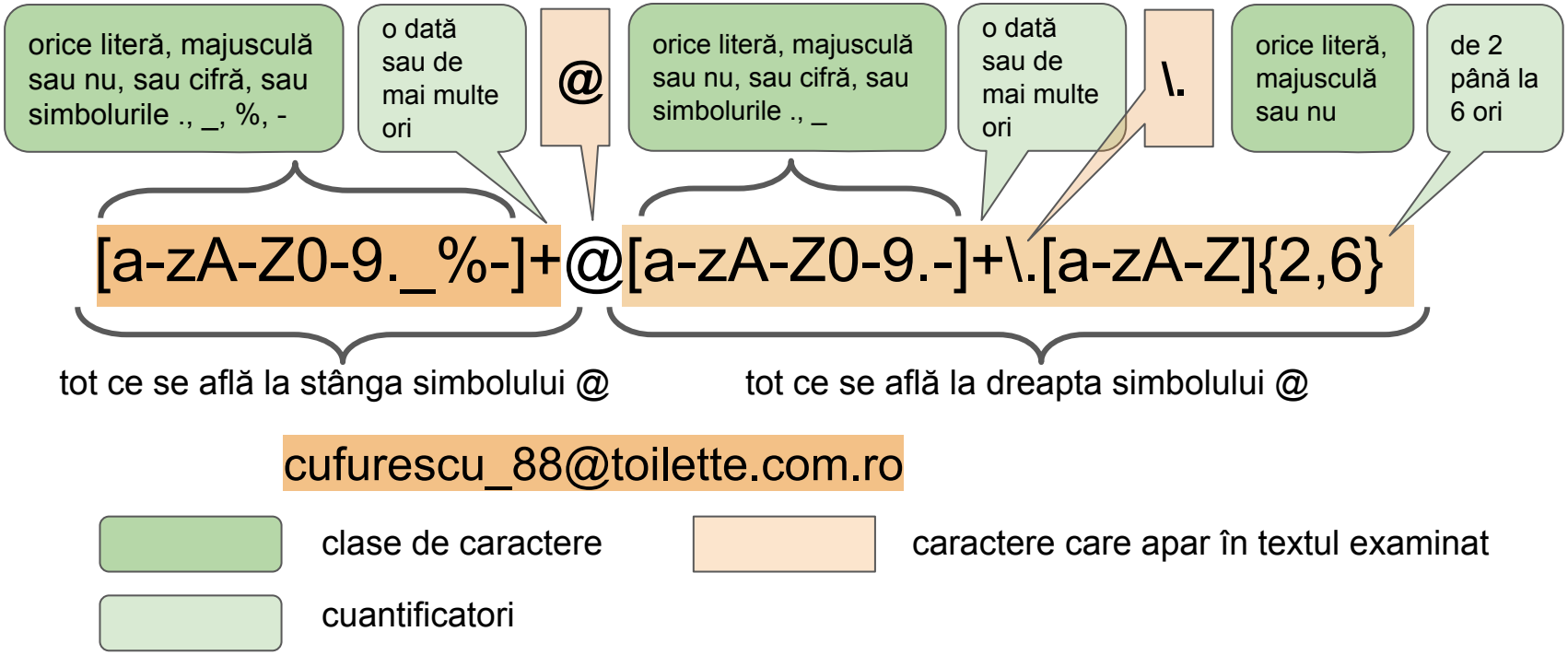
```
for (i = 0; i < persoane.length; i++) {
        print persoane[i].id, persoane[i].nume, persoane[i.prenume]
        for (j=0; j < orase.length; j++) {
                if (orase[j].id == persoane[i].oras) {
                        print orase[j].nume
                }
        }
}
```

SQL este cel mai utilizat limbaj pentru extragerea datelor structurate în modelul relational. Aproape orice aplicație cu care interacționați stochează datele într-o baza de date din care le extrage la solicitarea dvs.

Programare declarativă cu DSL - Expresii regulate

O expresie regulată este o colecție de caractere și simboluri care este interpretată ca un pattern de căutare. O extensie a conceptului de **caracter wildcard** .

Exemplu: expresie regulată care identifică adresele de e-mail.



cufurescu_88@toilette.com.ro

clase de caractere

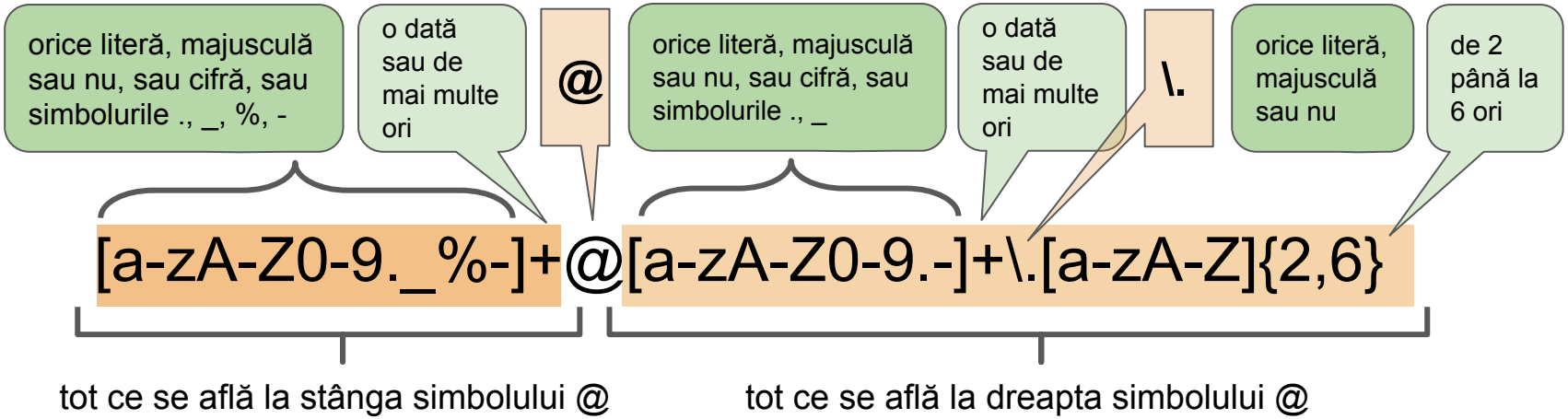
caractere care apar în textul examinat

cuantificatori

Caracterul . (punct) care trebuie să identifice punctul dintre domeniu și subdomeniu este precedat de un backslash (\) pentru că este și cuantificator în expresiile regulate. Acest procedeu se numește “escaping”.

În expresie există un singur punct (\.) deși în mod normal, in adresele de e-mail pot exista mai multe puncte nume@subdomeniu.domeniu.tld. Aceasta se întâmplă pentru că orice cuantificator dintr-o expresie regulată este lacom (greedy) - captează cel mai lung șir posibil.

Programare declarativă cu DSL - Expresii regulate



Cineva a scris un email de pe adresa cracanel@memyself.com către adresele: pompiliu.chininescu@doctorul.ro, cufurescu_88@toilette.com.ro în care menționa dorinta de a vinde un produs îndoielnic. Consider că acest abuz trebuie pedepsit cu tărie conform politicii de toleranță ZERO față de comunicări nesolicitate. Pentru aceasta, am trimis acest email si pe adresele: 1234567890@numarulcelmaimare.xxx, gentle-Man@iron-self.tv și Gladiather@caramela.tv. Dacă considerați necesar, vă rog sa trimiteți și către venerus@gineco.museum și carcalete@insecticid.co.jp și dacă e absolut necesar si către seful-cel-mai-mare@supercompanie.com.

Testare expresii regulate online:

https://regexr.com/

https://www.regextester.com/

ex_regexp_emails.php

https://onlinegdb.com/Hy36yAdDD

Programare declarativă cu DSL - Expresii regulate

un alt exemplu real - separarea unor autori dintr-o lista complexă

Giancane, G (Giancane, Gabriella)[1] ; Swart, JF (Swart, Joost F.)[2] ; Castagnola, E (Castagnola, Elio)[3] ; Groll, AH (Groll, Andreas H.)[4] ; Horneff, G (Horneff, Gerd)[5,6] ; Huppertz, HI (Huppertz, Hans-Iko)[7] ; Lovell, DJ (Lovell, Daniel J.)[8] ; Wolfs, T (Wolfs, Tom)[2] ; Herlin, T (Herlin, Troels)[9] ; Dolezalova, P (Dolezalova, Pavla)[10,11] ; Sanner, H (Sanner, Helga)[12,13] ; Susic, G (Susic, Gordana)[14] ; Sztajnbok, F (Sztajnbok, Flavio)[15] ; Maritsi, D (Maritsi, Despoina)[16] ; Constantin, T (Constantin, Tamas)[17] ; Vargova, V (Vargova, Veronika)[18] ; Sawhney, S (Sawhney, Sujata)[19] ; Rygg, M (Rygg, Marite)[20,21] ; Oliveira, SK (Oliveira, Sheila K.)[22] ; Cattalini, M (Cattalini, Marco)[23] ; Bovis, F (Bovis, Francesca)[1] ; Bagnasco, F (Bagnasco, Francesca)[1] ; Pistorio, A (Pistorio, Angela)[24] ; Martini, A (Martini, Alberto)[25] ; Wulffraat, N (Wulffraat, Nico)[2] ; Ruperto, N (Ruperto, Nicolino)[1]

```
( \[.*?\]\s; | \(.*?\) )
```

orice începe cu [și se termină cu] urmat de un spațiu si ; indiferent ce conține

sau

orice începe cu (și se termină cu) indiferent de ce conține

```
Giancane, G  Swart, JF  Castagnola, E  Groll, AH Horneff, G  Huppertz,
HI  Lovell, DJ  Wolfs, T Herlin, T  Dolezalova, P  Sanner, H  Susic, G
Sztajnbok, F  Maritsi, D  Constantin, T  Vargova, V Sawhney, S  Rygg,
M  Oliveira, SK  Cattalini, M Bovis, F  Bagnasco, F  Pistorio, A
Martini, A Wulffraat, N  Ruperto, N
```

https://onlinegdb.com/BkB9ZROvv

ex_regexp_authors.pl

Programare declarativă cu DSL - Expresii regulate

Lulu conduce bicicleta si masina; Judith conduce trotineta si vapor; Aladin conduce trotineta role si bicicleta; Heidi conduce avion role si bicicleta.

([A-Z]\w+)

Parantezele rotunde marchează un grup de captură

Fragmentul va putea fi folosit ulterior

Unul sau mai multe caractere de tip cuvânt

cuvântul conduce

cuvântul bicicletă

; sau .

```
([A-Z]\w+)\sconduce[^;]*?bicicleta.*?(;|\.)
```

Orice grup de litere care începe cu o majusculă

un spațiu

orice caracter care nu este ; de oricâte ori (*) până la prima apariție a ceea ce urmează (?)

orice grup de litere (.) de oricâte ori (*) până la prima apariție a ceea ce urmează (?)

Pentru rezolvarea acestei probleme, este necesară utilizarea unei expresii regulate de înlocuire, nu doar una de identificare.

De aceea folosim replace si indicam $1 ca fiind primul grup de captură.

```
var mytext = 'Lulu conduce bicicleta si masina; Judith conduce trotineta si vapor; Aladin conduce trotineta role si bicicleta; Heidi conduce avion role si bicicleta.';

while ($text =~ /([A-Z]\w+)\sconduce[^;]*?bicicleta.*?(;|\.)/g) {
   print $1."\n";
}
```

vehicule_regexp.pl

https://onlinegdb.com/rJyud1sOv

Programare declarativă cu DSL - HTML + CSS

```
<!DOCTYPE html>
  <html lang="ro-RO">
    <head>
      <meta charset="UTF-8" />
      <link rel="stylesheet" type="text/css" href="//www.utm.ro/wp-content/cache/wpfc-minified/10lvt9sj/1fmj.css"
media="all"/>
      <title>Universitatea Titu Maiorescu | Grad de încredere ridicat</title>
      <meta name="description" content="Facultăți: Psihologie, Drept, Medicină, Medicină Dentară, Farmacie, Științe Sociale,
Politice și Umaniste, Ştiinţe Economice şi Informatică." />
      <meta name="robots" content="index, follow" />
    </head>
    <body data-rsssl=1 class="home page-template-default page page-id-9 fusion-image-hovers fusion-pagination-sizing
wpb-js-composer js-comp-ver-6.4.1 vc_responsive">
      <a name="top" style="height:0; position:absolute; top:0;" id="top-anchor"></a>
      <div id="body-wrap">
        <div id="wrap">
          <header>
            <div id="top-nav" class="dark-div nav-style-2">
              <nav class="navbar navbar-inverse main-color-1-bg" role="navigation">
                <ul class="nav navbar-nav hidden-xs">
                 <li id="nav-menu-item-wpml-ls-5-ro" class="main-menu-item menu-item-depth-0 menu-item-language
menu-item-language-current menu-item wpml-ls-slot-5 wpml-ls-item wpml-ls-item-ro wpml-ls-current-language
wpml-ls-menu-item wpml-ls-first-item menu-item-type-wpml_ls_menu_item menu-item-object-wpml_ls_menu_item"><a
title="Română" href="https://www.utm.ro" class="menu-link  main-menu-link"><img onload="Wpfcll.r(this,true);"
src="https://www.utm.ro/wp-content/plugins/wp-fastest-cache-premium/pro/images/blank.gif" class="wpml-ls-flag"
data-wpfc-original-src="https://www.utm.ro/wp-content/plugins/sitepress-multilingual-cms/res/flags/en.png" alt="blank"><span
class="wpml-ls-native" lang="ro">Română</span> </a></li>
                 <li id="nav-menu-item-wpml-ls-5-en" class="main-menu-item menu-item-depth-0 menu-item-language
menu-item wpml-ls-slot-5 wpml-ls-item wpml-ls-item-en wpml-ls-menu-item wpml-ls-first-menu-item menu-item-type-wpml_ls_menu_item
menu-item-object-wpml_ls_menu_item"><a title="English" href="https://www.utm.ro/en" class="menu-link
main-menu-link"><img onload="Wpfcll.r(this,true);"
src="https://www.utm.ro/wp-content/plugins/wp-fastest-cache-premium/pro/images/blank.gif" class="wpml-ls-flag"
data-wpfc-original-src="https://www.utm.ro/wp-content/plugins/sitepress-multilingual-cms/res/flags/en.png" alt="blank"><span
class="wpml-ls-native" lang="en">English</span> </a></li>
                 </ul>
```

Header - nu este direct vizibil, conține instrucțiuni pentru motoarele de căutare, fișiere auxiliare, metadate.

DIV - Crează un container, ca un fel de partiție care este afișată conform specificațiilor aflate in proprietățile acestuia

UL - Crează o listă de elemente

HTML + CSS sunt declarative dar nu sunt chiar programare. DSL-ul nu este complet. Problema inversării listelor nu se poate rezolva în HTML + CSS acum. De aceea este nevoie încă de Javascript, pentru a suplimenta partea de programare.

Bicicliștii în HTML

```
<!--Contributia lui Ghermanescu Florin Adrian-->
<!-- se ruleaza prin incarcarea lui intr-un browser
(firefox, chrome) -->

<html>
<!--root tag pt inceperea documentului html-->
<style>
/* tag pt inceperea style-ului - se poate adauga si inline
*/
.vehicul {display: none;}
/* toate elementele apartinand clasei vehicul nu se vor
afisa */
.vehicul.bicicleta {display: block;}
/* suprascrierea declaratiei anterioare, astfel toate
elementele care apartin clasei vehicul SI clasei bicicleta
se vor afisa sub forma de block - selectorul este recunoscut
de la IE7*/
</style>
<!--tag pt incheierea style-ului-->
<p class="vehicul bicicleta masina">Lulu</p>
<!--html tag paragraph care atribuie textul "Lulu" claselor
vehicul, bicicleta si masina-->
<p class="vehicul trotineta vapor">Judith</p>
<p class="vehicul trotineta role bicicleta">Aladin</p>
<p class="vehicul avion role bicicleta">Heidi</p>
</html>
<!--root tag pt incheierea documentului html-->
```

Lulu

Aladin

Heidi

Combinația HTML - CSS determină vizibilitatea paragrafelor in funcție de clasa (<p>).

Se începe prin ascunderea tuturor paragrafelor cu ajutorul declaratiei CSS:

```
.vehicul {display: none;}
```

Paragrafele marcate cu clasa CSS “bicicleta” pot fi făcute vizibile:

```
.vehicul.bicicleta {display:
block;}
```

HTML + CSS sunt concepute pentru configurarea afișării - se poate scrie un exemplu care să afișeze conducătorii bicicletei dar fără interacțiune. Odată încărcat, documentul devine complet static.

Programare funcțională

Programarea funcțională este o paradigmă în care un program este compus prin **apelarea și compunerea funcțiilor**.

Din punct de vedere al codului, funcțiile sunt tot subrutine care au o serie de proprietăți suplimentare, care provin din faptul că sunt considerate "**componente de primă clasă** (first class citizens)" - elemente care suporta absolut toate operațiile disponibile. Astfel, față de o subrutină obișnuită, o funcție poate:

- să fie atribuită unui nume (ca o variabilă)
- să fie returnată dintr-o altă funcție
- să fie argument al unei funcții
- să poată fi verificată din punct de vedere al egalității

În limbajele de programare imperative, tipurile de date standard (int, etc.) sunt aproape întotdeauna componente de primă clasă.

În majoritatea limbajelor vechi (inclusiv C) tablourile NU sunt componente de primă clasă (de ex. în C nu pot fi trimise către o funcție sau returnate, aceste operații se realizează cu ajutorul pointerilor).

Programare funcțională pură - toate funcțiile NU au efecte secundare. Aceasta înseamnă că o funcție NU trebuie să modifice nimic în afara ei, nici o variabilă globală, sau trimisă prin referință. Astfel, starea generală a programului NU trebuie să se modifice în nici un fel în urma apelării unei funcții.

Programare funcțională (javascript)

```
const conducatori = [
        { nume: "Lulu", conduce: ["bicicleta", "masina"] },
        { nume: "Judith", conduce: ["trotineta", "vapor"] },
        { nume: "Alladin", conduce: ["trotineta", "role", "bicicleta"] },
        { nume: "Heidi", conduce: ["avion", "role", "bicicleta"] }];
```

Programare imperativă

```
let rezultate = [];
const cineconduce = "bicicleta";
for (i=0; i < conducatori.length; i++) {
 for (j=0; j < conducatori[i].conduce.length; j++) {
  if (conducatori[i].conduce[j] == cineconduce) {
   rezultate.push(conducatori[i].nume);
  }
 }
}
console.log(rezultate);
```

https://onlinegdb.com/HyMUUrJvv

Rhino nu suporta console.log, codul din spatele linkului foloseste print

Programare funcțională

```
const cineconduce = "bicicleta";

const rezultate = conducatori.filter(conducator =>
conducator.conduce.includes(cineconduce)).map(c
onducator => conducator.nume);

console.log(rezultate);
```

https://repl.it/languages/javascript

Rhino nu suportă instructiunile functionale

Limbaje de programare funcțională “pure” sunt: Haskel, Erlang

rezultate

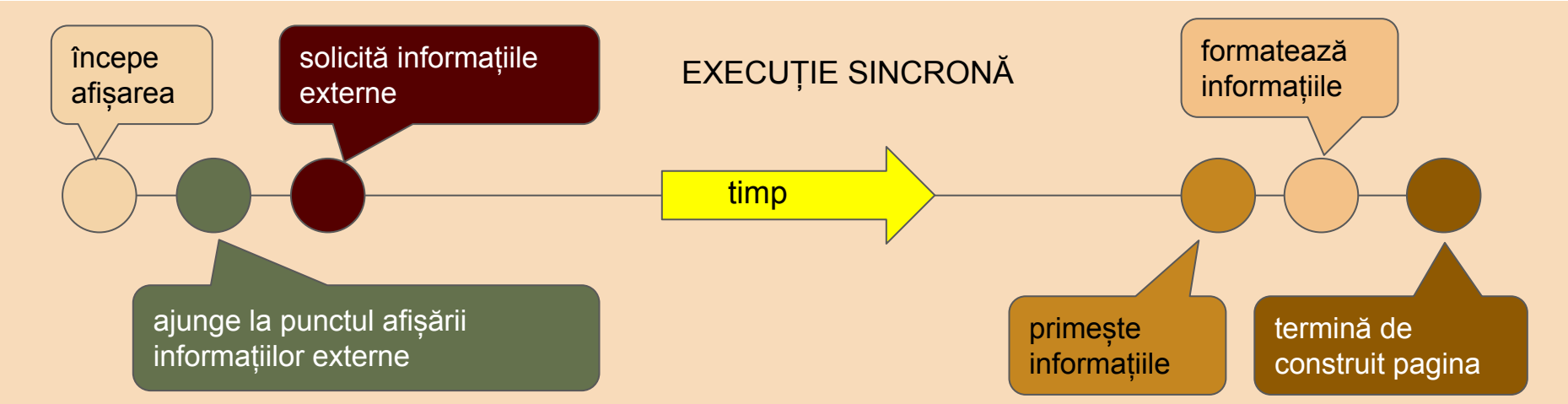
filter() includes() map()

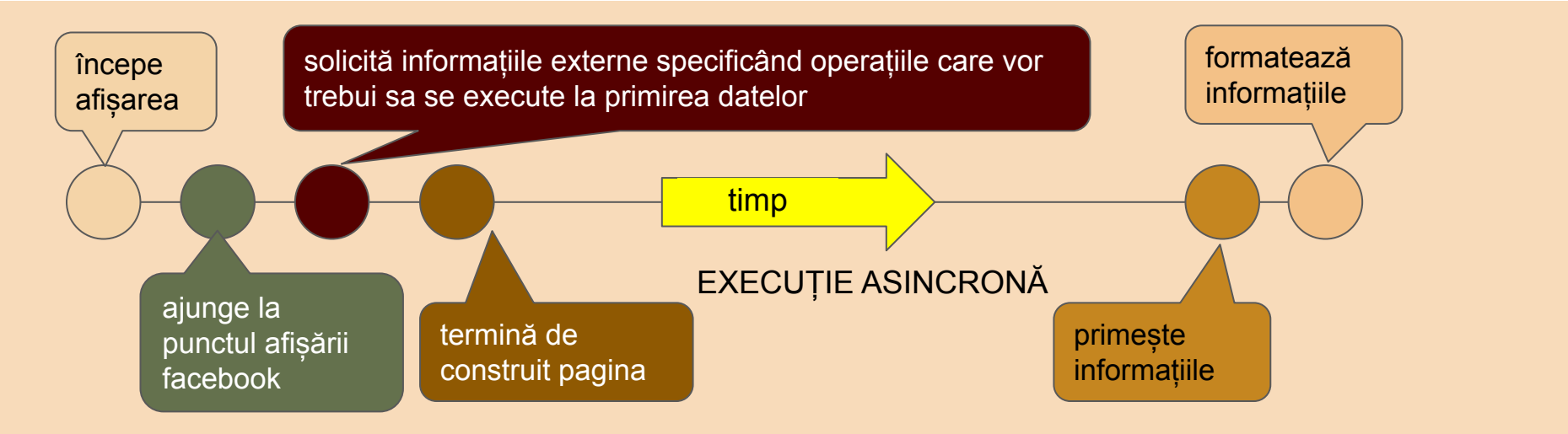
filter(includes()).map()

Javascript are multe elemente de programare funcțională. Dar, ca în orice domeniu, și în programare există obsesii majore pentru puritatea anumitor concepte. Există colective în care a numi Javascript “un limbaj funcțional” poate duce la consecințe grave.

Programare funcțională și asincronicitatea

Majoritatea programelor software sunt o combinație de cod care se poate executa foarte rapid și cod care se execută foarte lent.
Un exemplu simplu este un website care afișează (pe lângă conținutul pe care a fost programat să îl afișeze) un component pe care îl solicită de pe un serviciu de social media (ex: like-urile pe care le are pagina corespunzătoare pe facebook).





Asincronicitatea nu e o inventie nouă

Somon crocant cu legume

Peștele îl speli sub jet de apa rece și apoi îl condimentezi. Îl stropești cu zeama de lămâie și îl lași 15 minute.

Între timp, pregătești legumele.

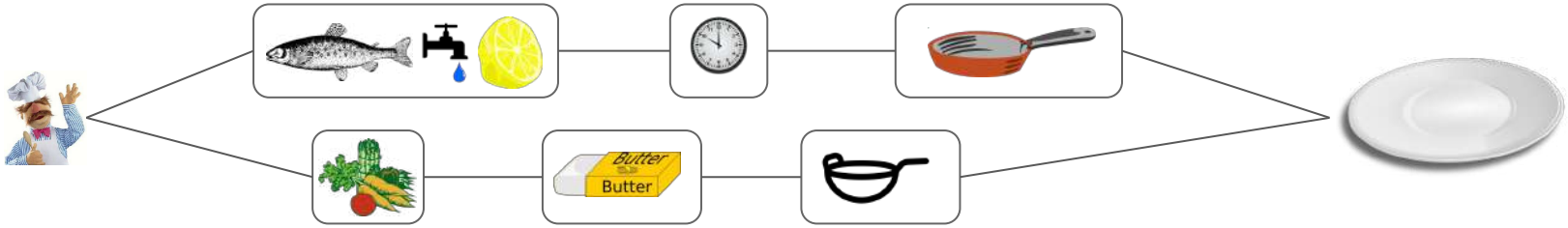
Într-o tigaie de teflon așezi bucățile de pește și le coci bine, câte 5 minute pe fiecare parte, până când devin aurii și ușor crocante.

Într-o tigaie wok pui untul la topit și adaugi legumele și apa, lăsându-le la foc mic până când aceasta scade. Condimentezi după gust.

Implementare :)

```
peste_spalat = Speli_pestele(“apa”: “rece”);
peste_condimentat = condimentare(peste_spalat);

peste_cu_lamaie = stropire(“subst”: lamaie,
callback: function() {
        asteapta_15_min(function(){
                // pestele ajunge in tigaie
                //pestele iese din tigaie
        })
});

legume_preparate = prepara_legumele()
… etc...
```

Timeline



Programare funcțională și asincronicitatea

```
// Solicită informa□ii despre server
xhrGET('/api/server-config', function(config) {
   // Solicită informa□ii despre user
   xhrGET('/api/' + config.USER END POINT, function(user) {
       // Solicită elementele apar□inând userului
       xhrGET('/api/' + user.id + '/items', function(items) {
           // Afi□ează elementele userului (e-mailuri, produse)
       });
   });
});
```

Instrucțiunea `xhrGET` primește ca argument url-ul și o funcție, numită callback pe care o va executa la primirea informațiilor.
După secvența de cod de mai sus, ceea ce urmează se poate executa înainte de venirea informațiilor de la cele trei apeluri.

```
xhrGET('/api/server-config', function(response) {
  server = response;
});
xhrGET('/api/' + server.USER_END_POINT, function(response) {
  user = response;
});
xhrGET('/api/' + user.id, function(response) {
  items = response;
});
for (i=0; i < items.length; i++) {
  //afișează items
}
```

Dacă se scrie codul secvențial, e posibil ca răspunsurile să nu vină în ordinea așteptată, caz în care apelurile succesive nu vor avea informațiile necesare.

Nevoia de programare asincrona a facut ca majoritatea limbajelor de programare sa capete elemente de programare functionala.

Programarea cu evenimente

O paradigma în care un program devine o listă completă de evenimente posibile cărora li se atașează funcții de tip callback care se execută când se declanșează evenimentul.

```
<conducatori>
  <conducator name="lulu">
    <vehicule>
      <vehicul>bicicleta</vehicul>
      <vehicul>masina</vehicul>
    </vehicule>
  </conducator>
  <conducator name="judith">
    <vehicule>
      <vehicul>trotineta</vehicul>
      <vehicul>vapor</vehicul>
    </vehicule>
  </conducator>
  <conducator name="alladin">
    <vehicule>
      <vehicul>trotineta</vehicul>
      <vehicul>role</vehicul>
      <vehicul>bicicleta</vehicul>
    </vehicule>
  </conducator>
  <conducator name="Heidi">
    <vehicule>
      <vehicul>avion</vehicul>
      <vehicul>role</vehicul>
      <vehicul>bicicleta</vehicul>
    </vehicule>
  </conducator>
</conducatori>
```

Document Object Model

DOM Parser - contruiește o structură de date arborescentă completă în memorie care conține întregul xml.

Fiecare element capătă o serie de metode de obținere a părintelui, a copiilor, a atributelor, etc.

children = node->children(); parent = node->parent(), etc.

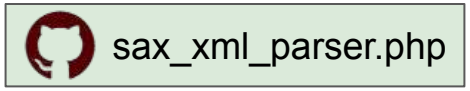
întreg arborele se încarcă în memorie

Simple Api for XML

SAX Parser - parcurge arborele fără să îl stocheze și declanșează câte un eveniment la fiecare nod.

Evenimentul este o funcție care va primi ca argument nodul respectiv și care decide ce face cu informația respectivă

Programarea cu evenimente - exemplu




```php
<?php

$conducator = null;

function startElement($parser, $name, $attrs) {
    global $conducator;
    if ($attrs && $attrs['NAME']) {
        $conducator=$attrs['NAME'];
    }
}

function endElement($parser, $name) { }

function continut($parser, $data) {
    global $conducator;
    if(!empty($data)) {
        if ($data == 'bicicleta') {
            echo ($conducator."\n");
        }
    }
}

$parser = xml_parser_create();

xml_set_element_handler($parser, "startElement", "endElement");
xml_set_character_data_handler($parser, "continut");

if (!($handle = fopen('conduce.xml', "r"))) {
    die("could not open XML input");
}

while($data = fread($handle, 4096)) {
    xml_parse($parser, $data);
}

xml_parser_free($parser);
```

Se executa la începutul oricărui tag

Se executa la finalul oricărui tag

Se execută la fiecare tag și primește conținutul acestuia

Specificarea componentelor parserului (functiile corespunzătoare)

Declanșează parsarea XML-ului

Event Loop

Limbajele interpretate (javascript, php, python, perl) sunt executate de un alt program (interpretor). în mod tradițional, un interpretor va citi programul instrucțiune cu instrucțiune și îl va executa instrucțiune cu instrucțiune.

În realitate, interpretorul poate primi comenzi proprii. Majoritatea interpretoarelor moderne sunt similare cu mașinile virtuale permițând “rearanjarea” execuției codului, împărțirea lui în mai multe fire de execuție, etc.

Un pattern de execuție frecvent folosit în programarea asincronă este “event loop” - un component care stochează o serie de operații care vor trebui executate la apariția unui eveniment sau arbitrar, când le vine rândul.

Intuitiv, orice interfață grafică este o colecție de “promisiuni” care se vor executa în cazul în care utilizatorul accesează unul dintre componentele acesteia.

Combinarea între event loop și programarea bazată pe evenimente permite execuția paralelă sau asincronă a funcțiilor declanșate de evenimente.

Cea mai cunoscută implementare a acestui concept este în motoarele de execuție javascript (atât cele din browsere cât și cel din node.js). Unul din motivele rapidității remarcabile a programelor executate cu ajutorul node.js este implementarea unui event loop care se bazează pe nucleul sistemului de operare.

Programarea logică

Un program este constituit dintr-o lista de axiome care se împart în fapte și clauze

Fiecare axioma este reprezentată printr-o clauză HORN

res :- cond1 și cond2 și cond3….

cond1 si cond 2 și cond3 implică res

rezultatul res se numește SCOP. El este adevărat dacă toate condițiile de după semnul :- (numite subscopuri) sunt adevărate.

Un program logic pornește de la o bază de cunoștințe care reprezintă o serie de propoziții adevărate exprimate sub forma clauzelor HORN.

Dacă baza de cunoștințe este completă și coerentă, programul poate să răspundă la întrebări derivate pe baza inferenței logice.

Programul în sine reprezintă un scop a cărui valoare de adevăr trebuie determinată.

Programarea logică - PROLOG

- Orice program prolog începe cu o listă de fapte numită “knowledge base”
- Lista de fapte este urmată de o listă de clauze - declarații logice care combină faptele cu variabile (variabilele joacă rolul predicatelor)
- Variabilele încep cu Majusculă iar constantele cu minusculă
- Declarațiile se termină cu un punct (.)

Knowledge base

```
tata(gomez,pugsley).
tata(gomez,wednesday).
mama(morticia,pugsley).
mama(morticia,wednesday).
```

Reguli complexe

```
frate(X,Y) :- mama(Z,X),
              mama(Z,Y),
              X \== Y.
```

```
?- mama(X,pugsley).
X=morticia
```

Cine este mama lui pugsley?
Ce trebuie sa pun în locul lui X astfel încât toată colecția de fapte să fie adevărată?

```
?- mama(morticia,Y).
Y=pugsley;
Y=wednesday
```

Cine sunt copiii morticiei?
Ce trebuie sa pun în locul lui Y astfel încât toată colecția de fapte să fie adevărată?

```
?- frate(X,Y).
X=pugsley
Y=wednesday
```

Care dintre ei sunt frați?
Ce trebuie sa pun în locul lui X și Y astfel încât clauza sibling să fie adevărată?

SCOP	frate(X,Y)
COND 1	mama(Z, X)
COND 2	mama(Z,Y)
COND 3	X \== Y

X și Y sunt frați dacă:

Z este mama lui X

Z este mama lui Y

X nu este același cu Y

family.pl

Programarea logică - Exemplu

Knowledge base

Clauza compusă

```
conduce(lulu, [bicicleta,masina]).
conduce(judith, [trotineta,vapor]).
conduce(aladin,[trotineta,role,bicicleta]).
conduce(heidi,[avion,role,bicicleta]).

cineconduce(Pers,Objct) :- conduce(Pers, Obiecte), member(Objct, Obiecte).
```

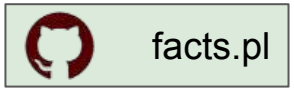
CLAUZĂ COMPUSĂ	cineconduce(Pers,Objct) :- conduce(Pers, Obiecte), member(Objct, Obiecte).

?- cineconduce(X,bicicleta).

X = lulu ;
X = aladin ;
X = heidi.

SCOP	cineconduce(Pers,Objct)	Persoana Pers conduce Objct dacă:
COND 1	conduce(Pers, Obiecte)	conduce Obiecte (căutare în liste)
COND 2	member(Objct, Obiecte)	din Obiectelface parte Objct

Întrebarea cineconduce(X,bicicleta) va parcurge toată baza de informații si va verifica veridicitatea clauzei compuse “cineconduce”. Va răspunde doar în cazul în care clauza compusă este adevărată.



Programarea logică - Utilizare specializată

Aplicații - orice program care trebuie sa construiască o configurație posibilă pe baza unui set de fapte existente. Foarte eficient la probleme care se rezolvă prin backtracking.

- Orarul unei facultăți.
- Planificarea călătoriilor, rutare.
- Partajarea resurselor comune
- Analiza limbajelor standardizate
- Sisteme decizionale
- Business intelligence

Datalog - subset al PROLOGULUI care este utilizat pentru baze de date (Datomic este un serviciu de baze de date distribuite care folosește datalog)

Programarea multiparadigmă (duh…)

Dezvoltarea instrumentelor de programare (nu doar compilatoare ci și întregul ansamblu de biblioteci pe care ajunge să le folosească un programator în dezvoltarea unei aplicații complexe) determină **tranziția dinspre imperativ spre declarativ.**

Multe sisteme pre-programate conțin componente flexibile, cu multe funcționalități, care pot fi apelate în programele noastre. Apelarea acestora este de cele mai multe ori **declarativă**.

Orice aplicație ceva mai complicată **va avea zone declarative semnificative** - fie încărcarea componentelor externe, fie interacțiunea cu sisteme de stocare (ex: SQL).

De asemenea, majoritatea aplicațiilor vor avea **zone funcționale**, fie reacționând la fluxul de date în stilul SAX, fie construind zone asincrone, sisteme înlănțuite de promisiuni, etc.

Orice demers serios de programare este multiparadigmă. Ceea ce este important este să detectați corect paradigma în care vă aflați în momentul respectiv. Dacă scrieți un sistem de configurare a unui component extern, nu căutați locul în care puteți sa scrieți instrucțiuni de ciclare, etc.

Recapitulare

Programarea calculatoarelor este una dintre cele mai recente concretizări a unei metode de **abstractizare și simplificare** a unor activități umane.

Programarea calculatoarelor este o **activitate umană** destinată unui procesor care lucrează **doar cu numere în baza 2**. Astfel, programarea trebuie în același timp:

- să permită oamenilor să opereze cu **concepte complexe**, așa cum sunt obișnuiți să o facă
- să trimită către procesor setul de instrucțiuni și date **simple** cu care acesta poate opera.

Astfel, limbajele de programare și compilatoarele trebuie să traducă o lume conceptuală aproximativă de complexitate aproape infinită intr-una extrem de simplă, aproape complet secvențială. Mai mult, traducerea trebuie făcută **fără pierdere de semnificație.**

Din acest motiv au apărut extrem de multe și de variate moduri de programare, împărțite în câteva grupuri mari numite **paradigme**.

Evoluția naturală a paradigmelor este o tranziție lentă **dinspre imperativ spre declarativ**. Cu cât apar mai multe instrumente gata construite, standardizate, grupate în sisteme și medii de lucru, cu atât programarea devine mai abstractă și mai declarativă.

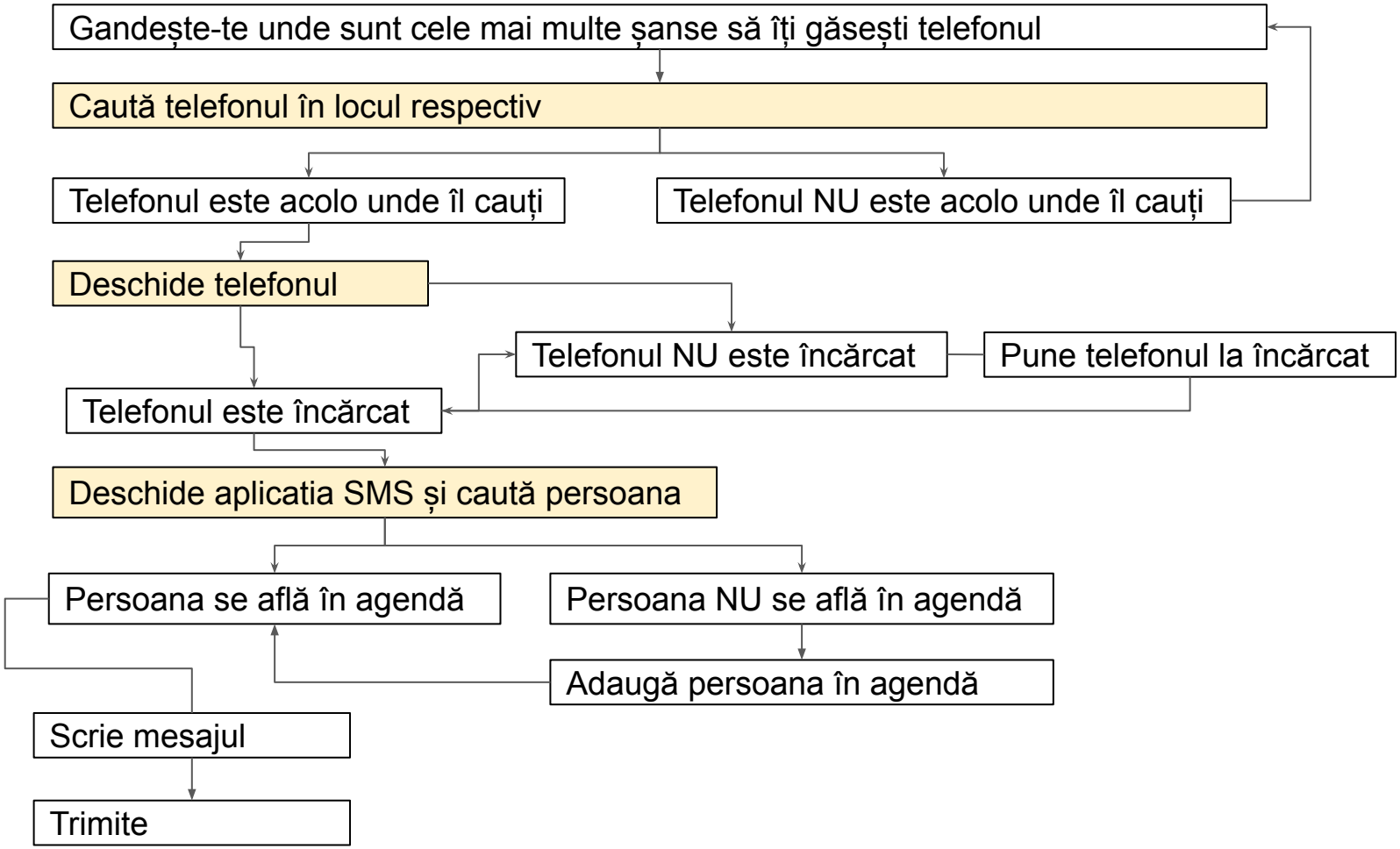
O alta tranziție perceptibilă este dinspre date puține și extrem de precis structurate către date extrem de multe și mai puțin structurate. Prelucrarea unei cantități foarte mari de date schimbă paradigma de programare - **fluxul de date devine elementul central** iar algoritmii sunt secundari.

5. Algoritmi, scheme logice și pseudocod

Algoritmi - trimiterea unui SMS

Un algoritm reprezintă un set clar de pași de rezolvare a unei probleme. Programele sunt implementări ale unuia sau mai multor algoritmi.

Aceeași problemă poate fi rezolvată prin mai mulți algoritmi diferiți.



Scheme logice - reprezentare a algoritmilor

Schemele logice sunt reprezentări grafice ale algoritmilor. Reprezentarea se face cu ajutorul unor diagrame care conțin casete standardizate unite prin legături.

Direcția și sensul execuției

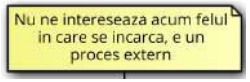
Început/sfârșit

Proces

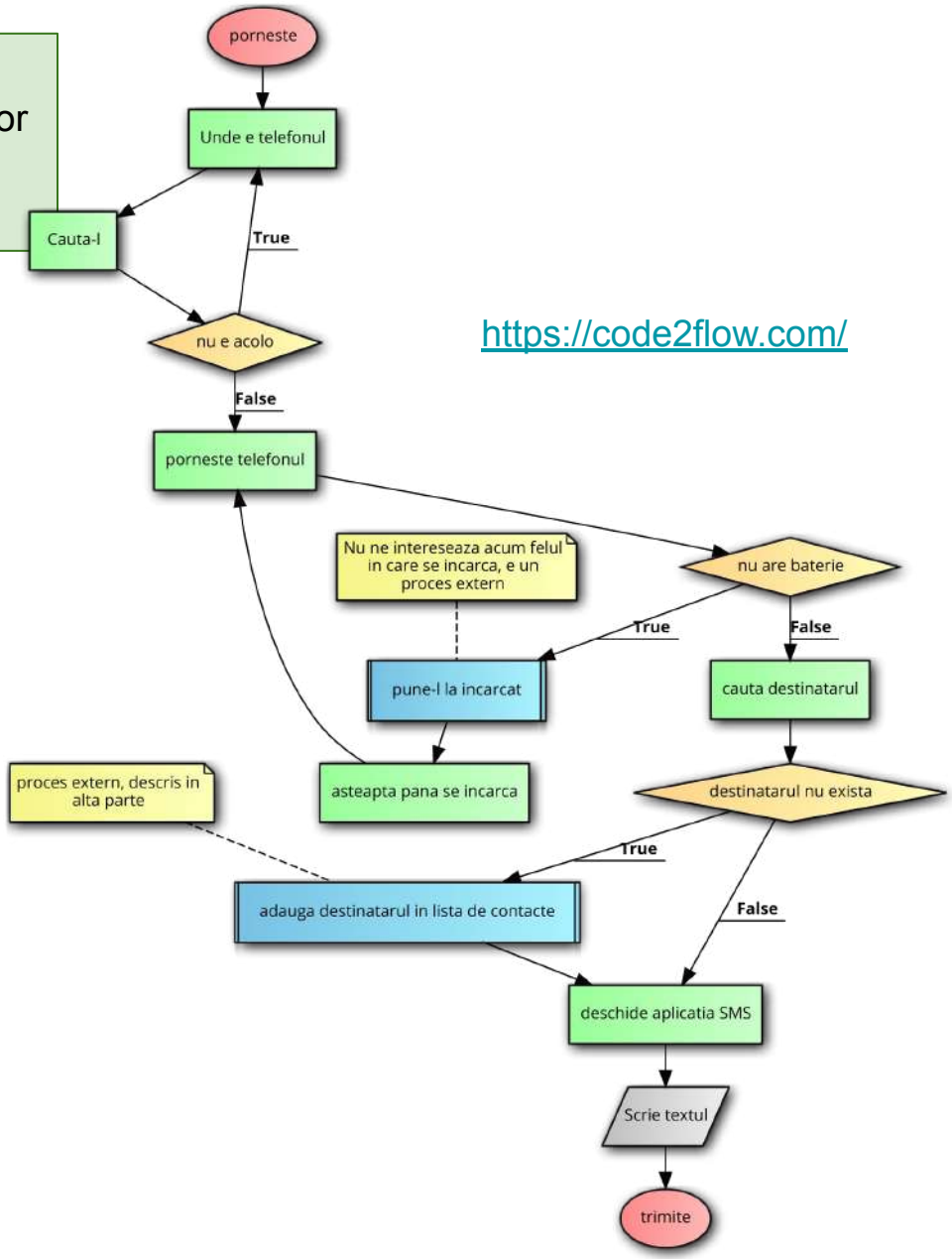
Condițional

I/O

Proces extern

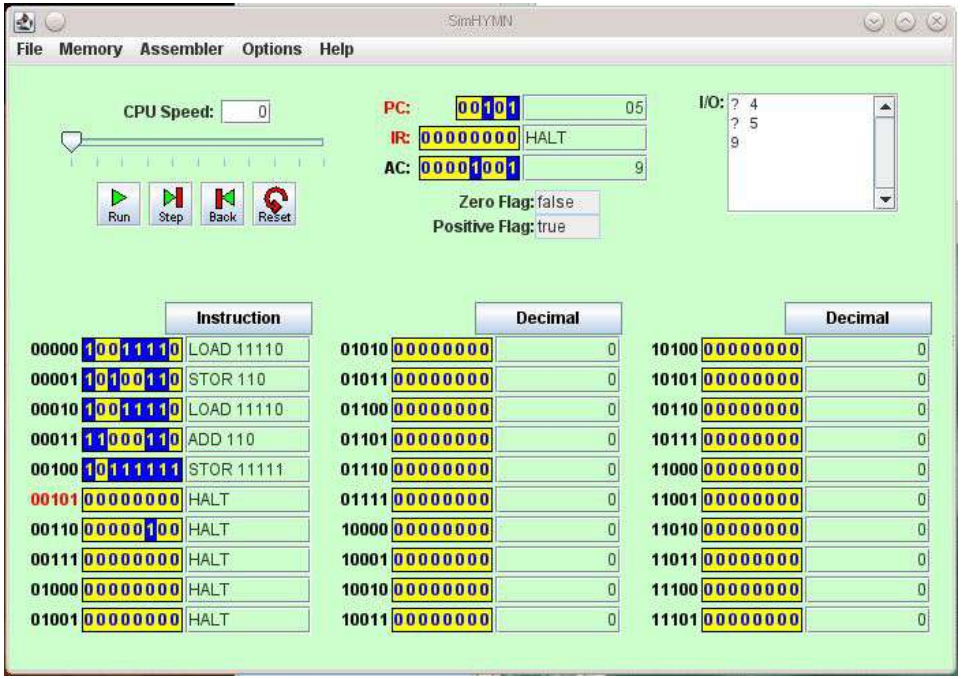


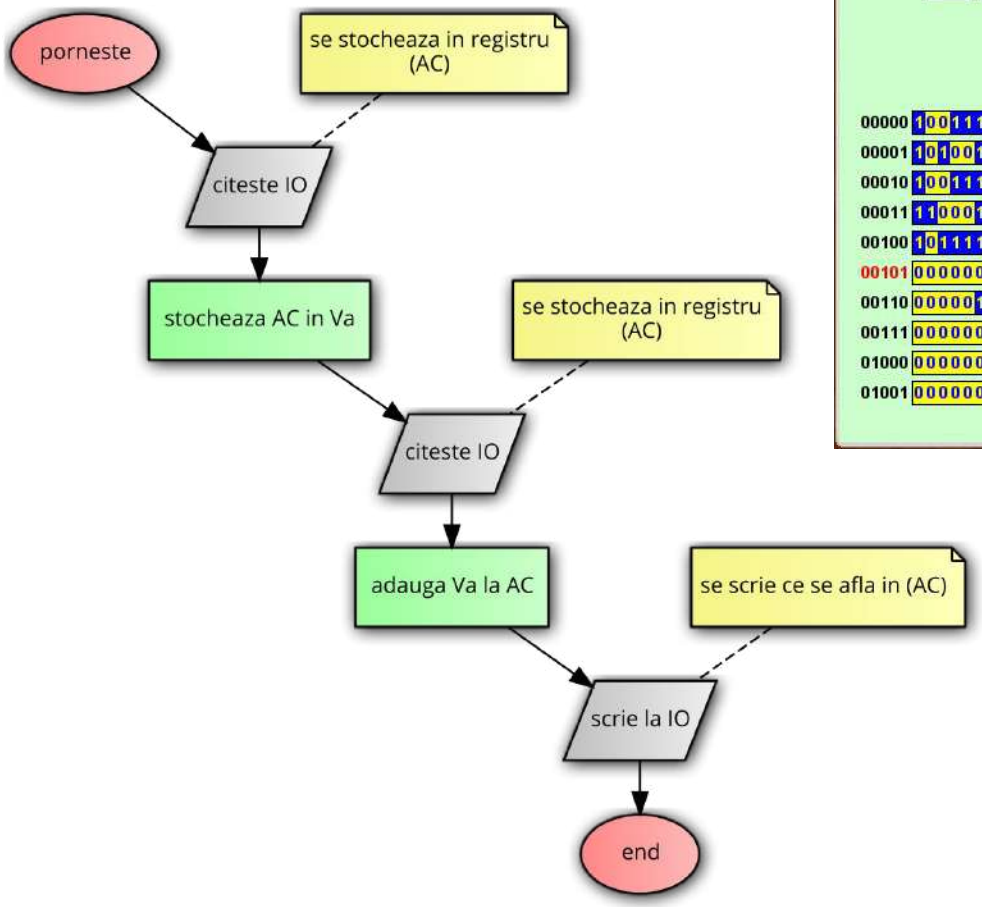
Comentariu



https://code2flow.com/

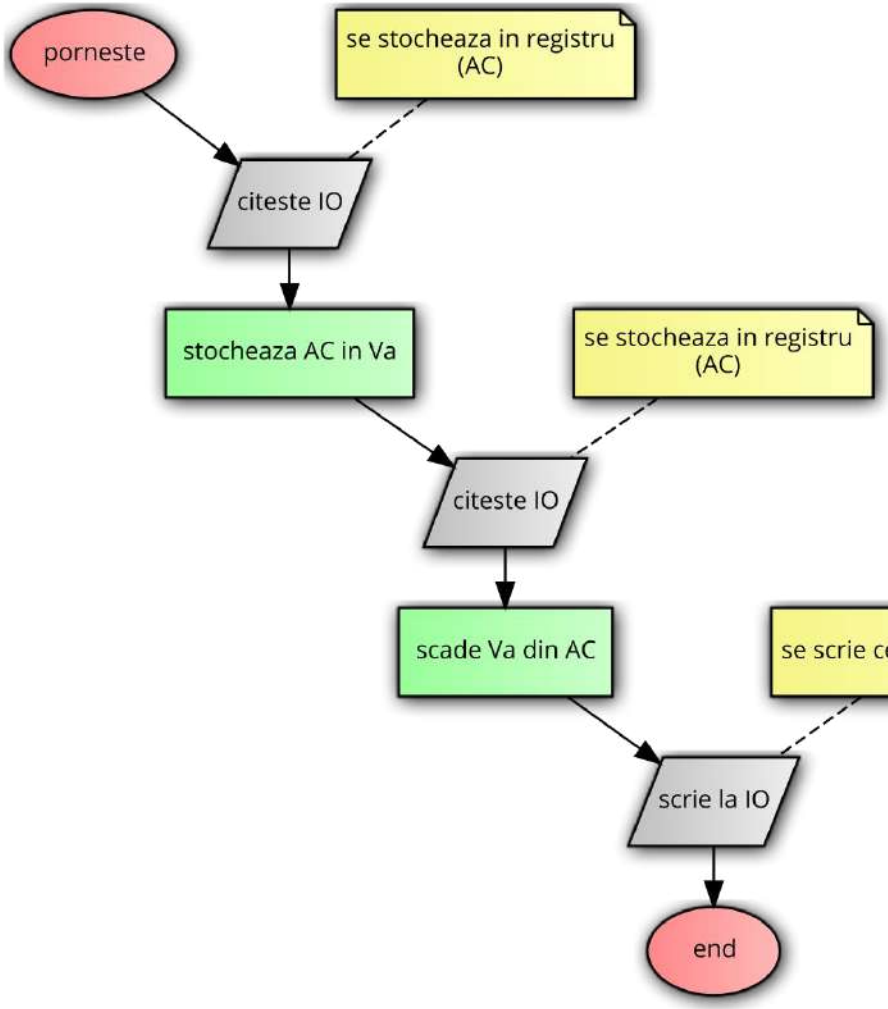
Adunarea a două numere întregi

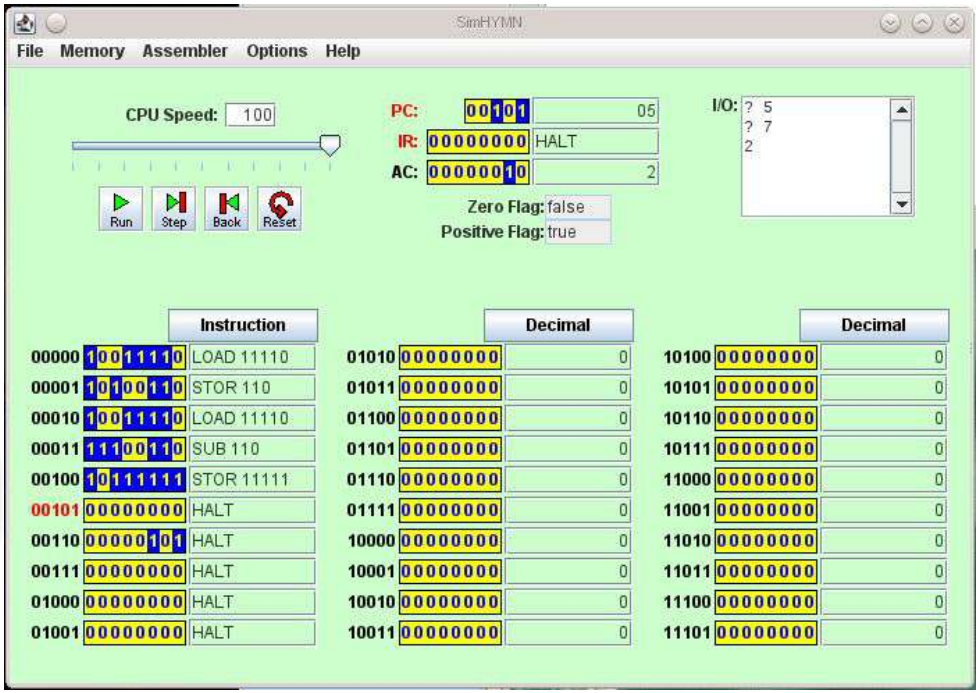





```
READ
STOR Va
READ
ADD Va
WRITE
HALT
Va:0
```

Scăderea a două numere întregi

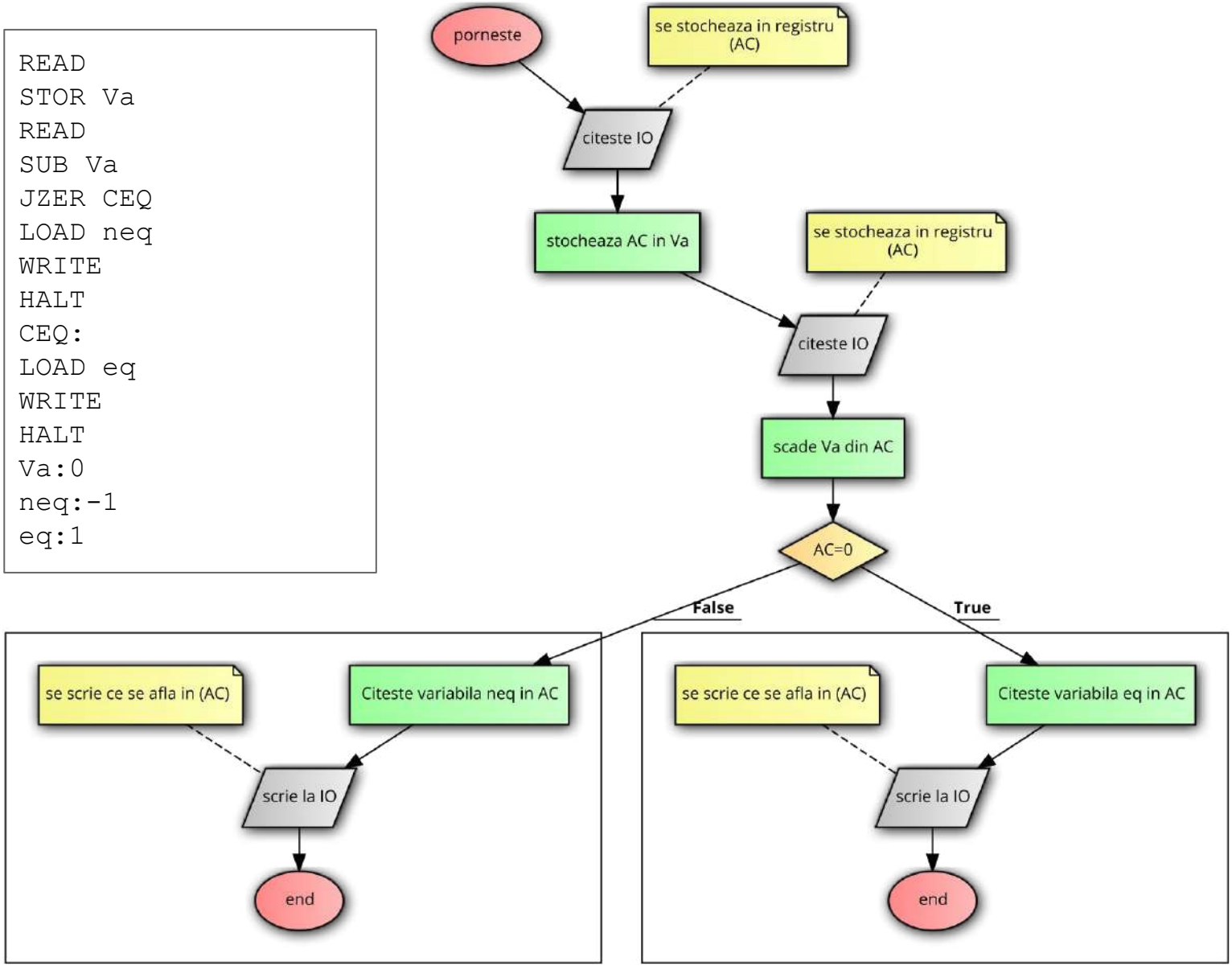





```
READ
STOR Va
READ
SUB Va
WRITE
HALT
Va:0
```

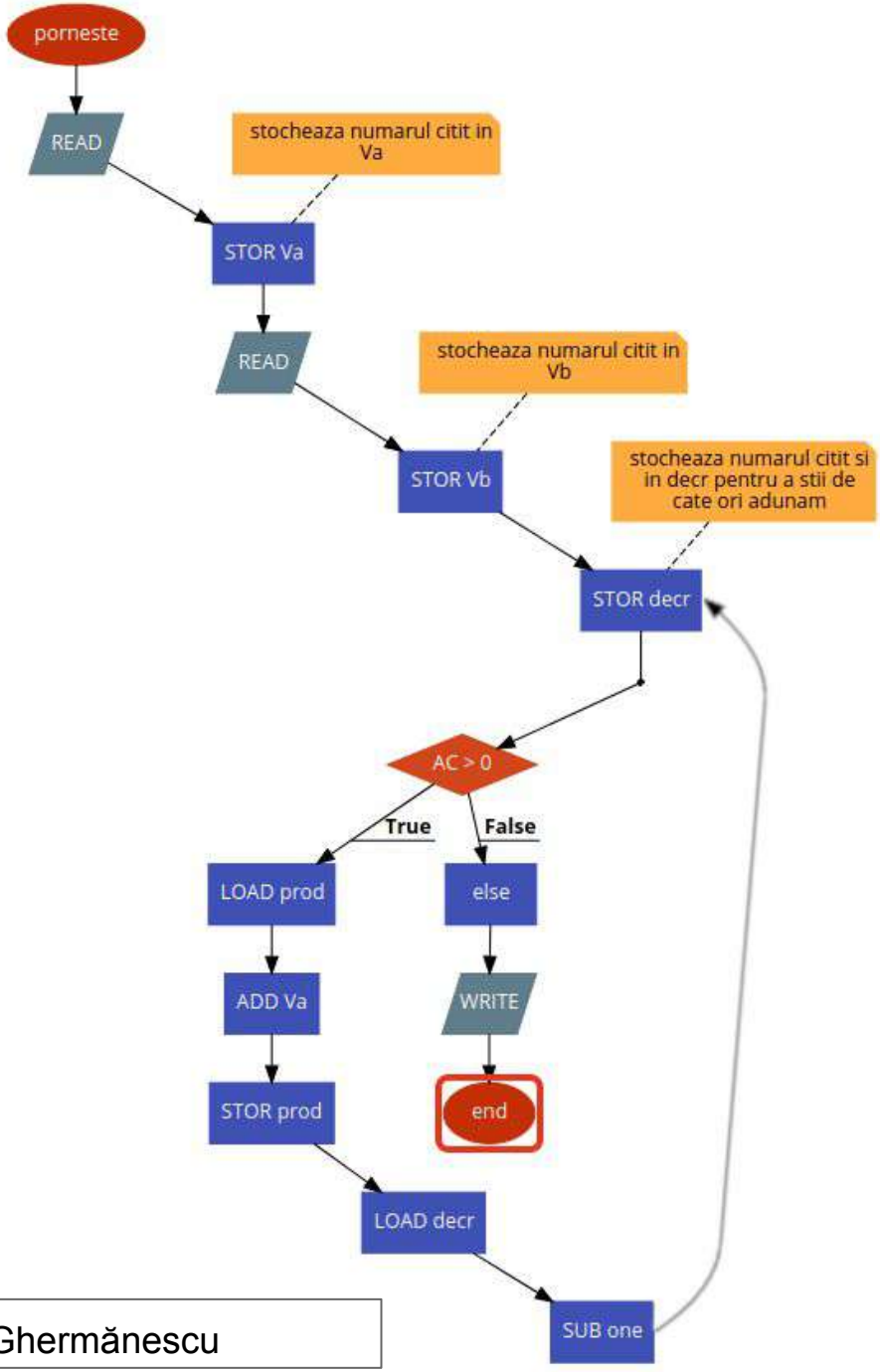
Compararea a două numere întregi

```
READ
STOR Va
READ
SUB Va
JZER CEQ
LOAD neq
WRITE
HALT
CEQ:
LOAD eq
WRITE
HALT
Va:0
neq:-1
eq:1
```



Înmulțirea a două numere întregi

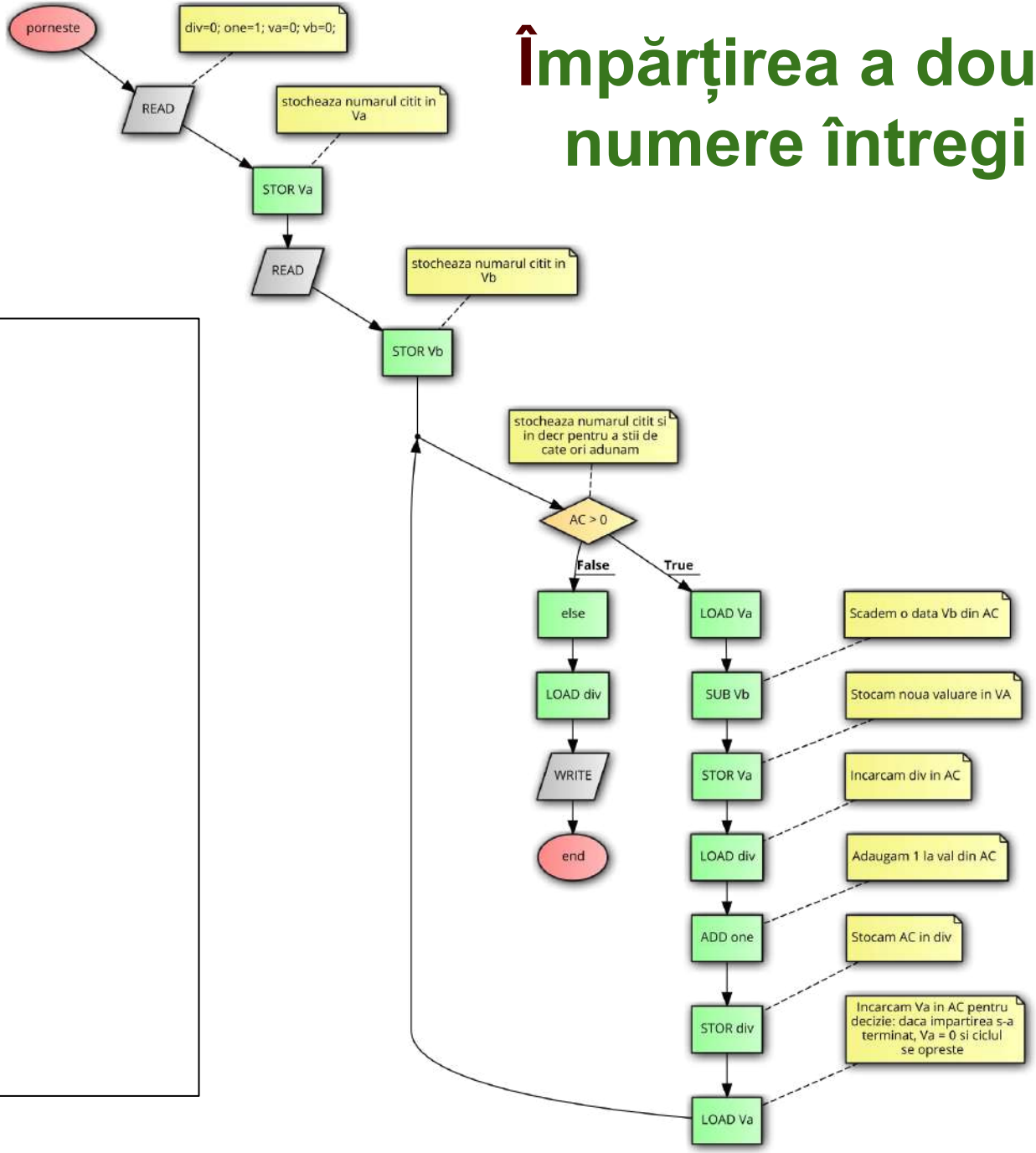
```
READ
STOR Va
READ
STOR Vb
STOR decr
LOOP:
   LOAD prod
   ADD Va
   STOR prod
   LOAD decr
   SUB one
   STOR decr
JPOS LOOP
LOAD prod
WRITE
HALT
Va:0
Vb:0
decr:0
one:1
prod:0
```



Simplificat cu ajutorul lui Florin Adrian Ghermănescu

Împărțirea a două numere întregi

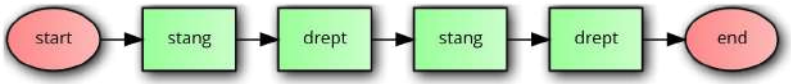
```
READ
STOR Va
READ
STOR Vb
LOOP:
        LOAD Va
        SUB Vb
        STOR Va
        LOAD div
        ADD one
        STOR div
        LOAD Va
JPOS LOOP
LOAD div
WRITE
HALT
Va:0
Vb:0
one:1
div:0
```



Structuri algoritmice

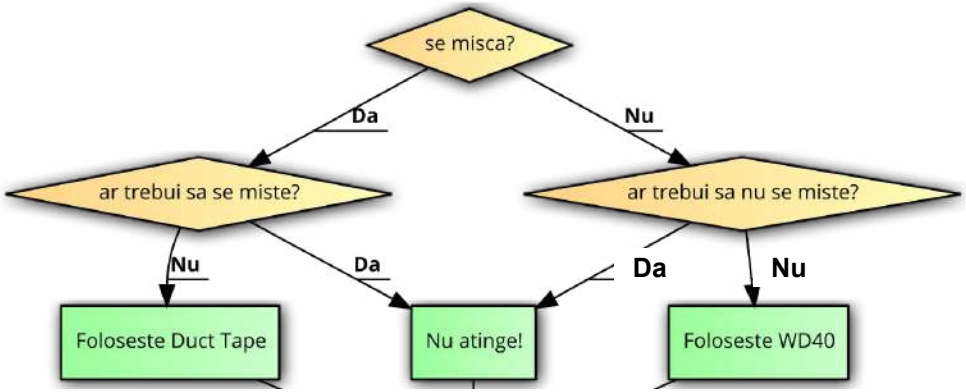
secvență

Secvența reprezintă o succesiune de pași care se execută în ordinea în care au fost scrișii în cod.



selecție

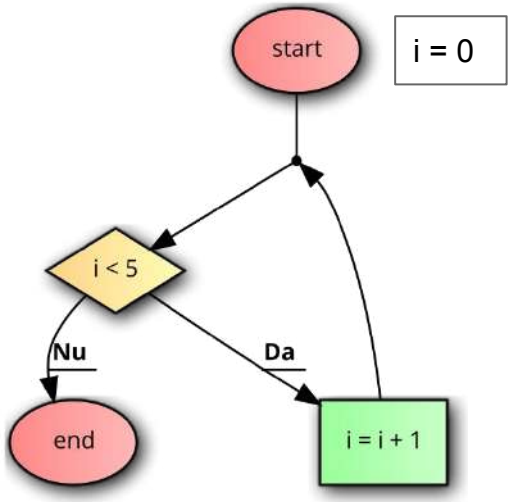
Selecția reprezintă o decizie care duce la execuția unor secvențe diferite în funcție de veridicitatea unei condiții predefinite.



If - then - elsif - else
switch - case

repetiție

Repetiția repetă o secvență o dată sau de mai multe ori până la îndeplinirea unei condiții predefinite.



for - end
do - while
while - end

Pseudocod

Pseudocodul este un mod de a descrie algoritmii care este foarte apropiat de un limbaj de programare. Scopul său nu este să fie compilat și executat de o mașină, ci să poată fi citit de alți oameni.

Evită declarațiile variabilelor (câteodată inițializarea lor), anumite funcții pot fi generalizate, anumiți operatori pot fi înlocuiți de expresii în limbaj natural.

Datorită formei sale libere, pseudocodul se poate scrie în multe feluri. De multe ori, pseudocodul se apropie de limbajul de programare folosit.

Stil FORTRAN

```
program sum100
  SET sumo = 0
  SET sume = 0
  Do i = 1 to 100
    IF i is odd
        sumo = sumo + i
    ENDIF
    IF i is even
        sume = sume - i
    ENDIF
    PRINT sumo
    PRINT sume
  end do
```

Stil PASCAL

```
procedure sum100
  sumo := 0
  sume := 0
  for i := 1 to 100 do
    if i is odd then
        sumo := sumo + i
    end
    if i is even then
        sume := sume - i
    end
    print sumo
    print sume
end
```

Stil C

```
void function sum100 {
  sumo = 0
  sume = 0
  for (i = 1; i <= 100; i++) {
    if i is odd {
      sumo = sumo + i
    }
    if i is even {
      sume = sume + 1
    }
    print sumo
    print sume
  }
}
```

Pseudocod - exemple

media mai multor numere

```
read n, N1,N2...Nn
suma = 0
contor = 1
repeat n times
     sum = sum + ni
     i = i + 1
end repeat
average = sum / i
write average
```

media a două numere

```
read a, b
average = a + b / 2
write average
```

media mai multor numere

```
read n, N1,N2...Nn
suma = 0
cnt = 1
for cnt = 1, n, 1
     sum = sum + Ncnt
     ~~i = i + 1~~
end for
average = sum / i
write average
```

sau

media mai multor numere

```
read n, n1,n2...nn
suma = 0
contor = 1
while i < n
     sum = sum + ni
     i = i + 1
end while
average = sum / i
write average
```

media mai multor numere

```
read n, n1,n2...nn
suma = 0
contor = 1
do
     sum = sum + ni
     i = i + 1
while i < n
average = sum / i
write average
```

SECVENȚE

Secvența reprezintă o succesiune de pași care se execută în ordinea în care au fost scriși în cod.

```
a = 12
b = x + y / z
write "ceva"
read N
```

```
write "principal_amt, rate and time"
read principal_amt, rate, time
interest = (principal_amt * rate * time) / 100.0
write "Amount", principal_amt
write "Rate", rate
write "Time", time
write "Interest, interest
```

SELECȚIE: if - then - else

pseudocod

```
write "give me N"
read N
if N > 0 then
    write "Numarul e mai mare decat zero"
else
    write "Numarul e mai mic decat zero"
end if
```

Selecția reprezintă o decizie care duce la execuția unor secvențe diferite în funcție de veridicitatea unei condiții predefinite.

c

if_else.c

```
#include <stdio.h>

int main()
{
    int N;
    printf("introduceti un numar\n");
    scanf("%i",&N);
    if (N > 0) {
        printf("Numarul e mai mare decat zero\n");
    } else {
        printf("Numarul e mai mic decat zero\n");
    }
    return 0;
}
```

https://onlinegdb.com/Hy_RFIN6m

SELECȚIE: if - then - else if - else

pseudocod

```
write "introduceti N"
read N
if N > 0 then
    write "Numarul e mai mare decat zero"
else if N == then 0
    write "Numarul e zero"
else
    write "Numarul e mai mic decat zero"
end if
```

Selecția reprezintă o decizie care duce la execuția unor secvențe diferite în funcție de veridicitatea unei condiții predefinite.

c

if_else_zero.c

```c
#include <stdio.h>
int main()
{
    int N;
    printf("give me a Number\n");
    scanf("%i",&N);
    if (N > 0) {
        printf("Numarul e mai mare decat zero\n");
    } else if (N == 0) {
        printf("Numarul e zero\n");
    } else {
        printf("Numarul e mai mic decat zero\n");
    }
    return 0;
}
```

https://onlinegdb.com/BJSuqIN6m

SELECȚIE: switch - case

Selecția reprezintă o decizie care duce la execuția unor secvențe diferite în funcție de veridicitatea unei condiții predefinite.

pseudocod

```
char gr = 'B';
case on grade
  case 'A' :
    write "Excelent!", gr
    break;
  case 'B' :
    write "Close, but no cigar", gr
    break;
  case 'C' :
    write "On the top of gauss hat", gr
    break;
  case 'D' :
    write "Still in school, not for long", gr
    break;
  case 'F' :
    write "you need a job\n", gr
    break;
  default :
    write "Invalid grade", gr
  end case
```

c

switch.c

```c
#include <stdio.h>

int main () {
  char gr = 'B';
  switch(gr) {
     case 'A' :
        printf("%c: Excelent!\n", gr );
        break;
     case 'B' :
        printf("%c: Close, but no cigar\n", gr );
        break;
     case 'C' :
        printf("%c: On the top of gauss hat\n", gr);
        break;
     case 'D' :
        printf("%c: Still in school, not for long\n",
gr);
        break;
     case 'F' :
        printf("%c: you need a job\n", gr);
        break;
     default :
        printf("%c: Invalid grade\n", gr);
  }
  return 0;
}
```

https://onlinegdb.com/r1XEOINa7

```
B: Close, but no cigar
```

SELECȚIE: switch - case fără break

pseudocod

```
char gr = 'B';
Case on grade {
  case 'A' :
    write "Excelent!", gr
  case 'B' :
    write "Close, but no cigar", gr
  case 'C' :
    write "On the top of gauss hat", gr
  case 'D' :
    write "Still in school, not for long", gr
  case 'F' :
    write "you need a job\n", gr
  default :
    write "Invalid grade", gr
 }
```

c

switch-nobreak.c

```c
#include <stdio.h>

int main () {
  char gr = 'B';
  switch(gr) {
     case 'A' :
        printf("%c: Excelent!\n", gr );
     case 'B' :
        printf("%c: Close, but no cigar\n", gr );
     case 'C' :
        printf("%c: On the top of gauss hat\n", gr);
     case 'D' :
        printf("%c: Still in school, not for long\n",
gr);
     case 'F' :
        printf("%c: you need a job\n", gr);
     default :
        printf("%c: Invalid grade\n", gr);
  }
  return 0;
}
```

Break-ul este necesar în cazul switch-ului

https://onlinegdb.com/rkaSO8VaQ

```
B: Close, but no cigar
B: On the top of gauss hat
B: Still in school, not for long
B: you need a job
B: Invalid grade
```

REPETIȚIE: media

bucla while

pseudocod

```
suma = 0
cnt = 1
avg = 0
write "give me N"
read N
while cnt <= N
    write "Numarul nr.", cnt
    read tmp
    sum = sum + tmp
    avg = sum / cnt
    cnt = cnt + 1
end while
average = sum / i
write "media este: ", average
```

c

average-while.c

```c
#include <stdio.h>

int main()
{
    int sum=0, count=1, N=0, tmp=0;
    float avg;
    printf("give me N\n");
    scanf("%i",&N);
    while (count < N+1) {
        printf("Numarul nr. %i\n", count);
        scanf ("%i",&tmp);
        sum = sum + tmp;
        avg = (float)sum / count;
        count = count+1;
    }

    printf("media este: %f\n",avg);
    return 0;
}
```

https://onlinegdb.com/BkfimULPP

```
while (cnt <= N) {
    ….
}
```

```
give me N
3
Numarul nr. 1
12
Numarul nr. 2
17
Numarul nr. 3
2
media este: 10.333333
```

Buclele de tip while sunt bucle care se executa atâta timp cât condiția din paranteză este adevărată. Verificarea se face înainte de execuția blocului de cod din bucla.

REPETIȚIE: Calculul mediei

bucla do

pseudocod

```
suma = 0
cnt = 1
avg = 0
write "give me N"
read N
do
    write "Numarul nr.", cnt
    read tmp
    sum = sum + tmp
    avg = sum / cnt
    cnt = cnt + 1
while count <= N

write "media este: ", average
```

c

average-do.c

```c
#include <stdio.h>

int main()
{
    int sum, N, tmp;
    int count = 1;
    float avg;
    printf("give me N\n");
    scanf("%i",&N);
    do {
        printf("Numarul nr. %i\n", count);
        scanf ("%i",&tmp);
        sum = sum + tmp;
        avg = (float)sum / count;
        count = count+1;
    } while (count <= N);

    printf("media este: %f\n",avg);
    return 0;
}
```

```
do {
    ….
}while (cnt <= N);
```

https://onlinegdb.com/B12kKLEa7

Buclele de tip do - while sunt bucle care se executa atâta timp cât condiția din paranteză este adevărată. Verificarea se face după o execuție a blocului de cod din bucla.

REPETIȚIE: Calculul mediei

bucla for

pseudocod

```
suma = 0
cnt = 1
avg = 0
write "give me N"
read N
for cnt = 1, N, 1
    write "Numarul nr.", cnt
    read tmp
    sum = sum + tmp
    avg = sum / cnt
end for

write "media este: ", average
```

c

average-for.c

```c
#include <stdio.h>

int main()
{
    int sum, count, N, tmp;
    float avg;
    printf("give me N\n");
    scanf("%i",&N);
    for (count = 1; count <= N; count++) {
        printf("Numarul nr. %i\n", count);
        scanf ("%i",&tmp);
        sum = sum + tmp;
        avg = (float)sum / count;
    }

    printf("media este: %f\n",avg);
    return 0;
}
```

```
for (cnt = 1; cnt <= N; cnt++) {
    ….
}
```

https://onlinegdb.com/rk6qFI4pX

Buclele de tip for sunt bucle la care numărul de iterații este declarat în momentul execuției buclei.

În pseudocod acestea se pot scrie și sub forma repeat N

pseudocod - recapitulare

secvență

```
a = 12
b = x + y / z
write "ceva"
read N
```

selecție

```
if N > 0 then
     ...
else
     ...
end if
```

```
if N > 0 then
     ...
else if N == 0 then
     ...
else
     ...
end if
```

```
case on grade
  'A'
     ...
     break
  'B'
     ...
    break
  default :
     ...
  end case
```

repetiție

```
while cnt <= N
    …
end while
```

```
do
    ...
while count <= N
```

```
for cnt = 1, N, 1
    ...
end for
```

Utilitatea pseudocodului

Pseudocodul este util în special în proiectare. Mai ales când există echipe de programatori mixte.

În funcție de obișnuință poate fi mai rapidă scrierea unui algoritm în pseudocod decât într-un limbaj de programare (care adaugă declaratii de variabile, etc.)

Cel mai prost moment pentru utilizarea pseudocodului este în timpul învățării unui limbaj de programare.

6. Funcții, proceduri, subrutine

Funcții, proceduri, subrutine

Procedurile reprezintă **mici subprograme** care pot fi separate de corpul programului principal. Ce anume intră într-o procedură reprezintă o decizie de proiectare a programului. În anumite limbaje de programare acestea sunt numite proceduri, în altele funcții, în altele subrutine.

C nu are un cuvânt special pentru definirea procedurilor, acestea sunt specificate prin nume, argumente precum și prin tipul datei pe care îl returnează. Mai multe limbaje de programare urmează același model (Java, C#, Python, Ruby, Scala). Alte limbaje folosesc cuvinte rezervate (instrucțiuni) pentru declararea unei proceduri.

Javascript	function	function solveWorldHunger(a, b, c)
Pascal	procedure	procedure solveWorldHunger(a, b, c: integer;)
Basic	sub	sub solveWorldHunger(a,b,c)
Php	function	function solveWorldHunger(a, b, c)
Fortran	subroutine	subroutine solveWorldHunger(a,b,c)
Perl	sub	sub solveWorldHunger
Lisp	defun	defun solveWorldHunger(a,b,c)

Funcții, proceduri, subrutine

Orice limbaj de programare de nivel înalt are o serie de funcții predefinite care pot fi utilizate direct în cod.

Orice limbaj de programare permite definirea propriilor funcții.

Împărțirea unui program în funcții ajută la:

1. Ușurința înțelegerii codului de către oameni
2. Rapiditatea scrierii codului în cazul operațiilor repetitive (ex: calculul radicalului unui număr, care poate apărea de foarte multe ori într-un program).
3. Depanarea mai ușoară - în cazul în care există o greșeală într-o funcție, ea trebuie corectată într-un singur loc.
4. Reutilizarea codului - funcțiile pot fi copiate dintr-un program în altul sau chiar grupate în biblioteci care pot fi refolosite.
5. Posibilitatea proiectării în etape a unui program complex.
6. Executia algoritmilor recursivi

Toate fragmentele de cod C afișate mai sus încep cu linia `#include <stdio.h>`. Această linie anunță compilatorul că avem intenția de a utiliza funcții definite în biblioteca **stdio**, bibliotecă ce însoțește compilatoarele C.

Proceduri, subrutine - suport hardware

Programarea procedurală implică utilizarea procedurilor pentru a **separa** fragmente de cod care pot fi izolate. Rezultă un cod modular care este **mai ușor de citit**, **mai ușor de depanat**, de multe ori **mai scurt** (sunt multe fragmente de cod care se execută de multe ori).

Uneori există colecții de proceduri care se utilizează des și care sunt caracterizate de intrări și ieșiri foarte clare. Acestea sunt de obicei grupate în **biblioteci (libraries)** care sunt utilizate în timpul programării.

Principala provocare pusă în fața unui programator nu este aproape niciodată sintaxa limbajului de programare pe care îl folosește ci **cunoașterea operațională a colecției de biblioteci** pe care trebuie să le utilizeze.

In cazul C, biblioteci comune sunt: stdio, string, time, math, limits, threads, stdlib, ctype

Programarea în limbaj de asamblare poate fi procedurală. Prin existența instrucțiunilor de tip JUMP, diferite fragmente de cod pot fi poziționate în memorie la adrese precise iar procesorul poate fi instruit să “sară” de la una la alta.

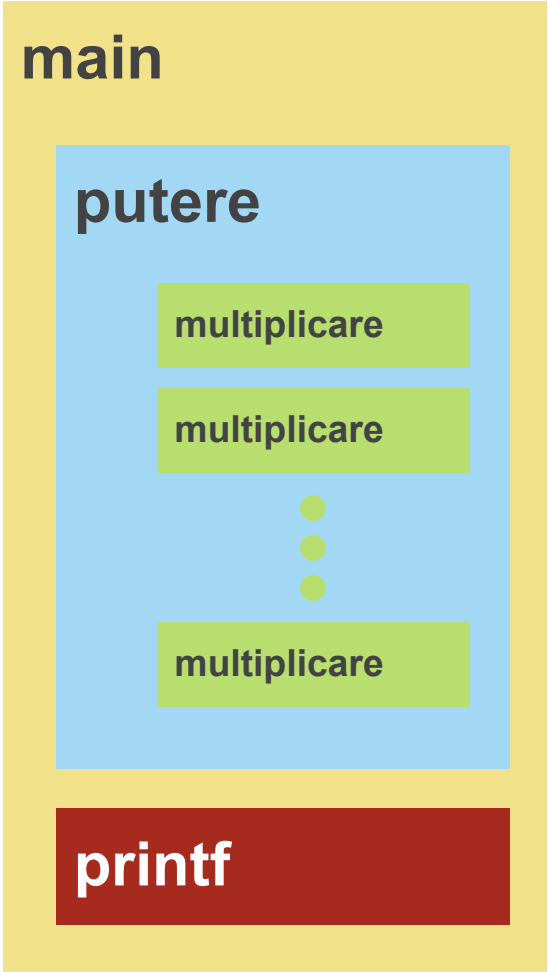
În plus, registrele procesorului pot fi utilizate pentru a reține adresa procedurii care a apelat procedura curentă, permițând reîntoarcerea.

Prin combinația utilizării registrelor, a instrucțiunilor de salt și a modului de acces al memoriei, compilatoarele pot crea un model special de execuție pentru programele procedurale.

Programarea procedurală are suport hardware!

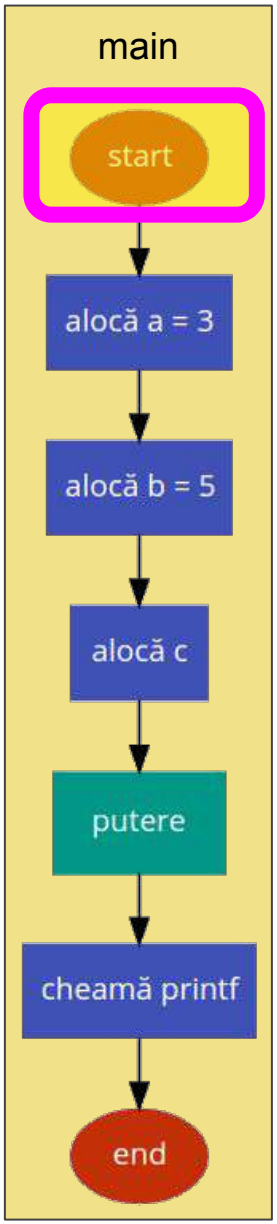
Viața unui program

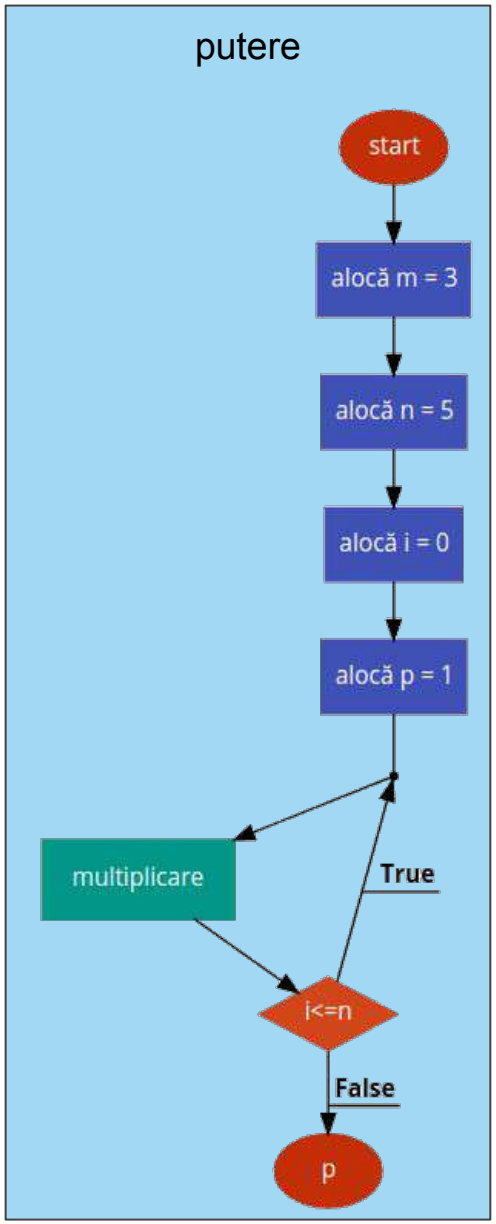
```
#include <stdio.h>

int multiplicare(int m, int n) {
   int result = 0;
   result = m * n;
   return result;
}

int putere(int m, int n)
{
  int i;
  int p=1;
  for (i=1; i<=n; i++) {
      p = multiplicare(p,m);
  }
  return p;
}

int main(int argc, char *argv[])
{
  int a=3;
  int b=5;
  int c = putere(a, b);
  printf("%d la puterea %d este %d\n",a,b,c);
  return 0;
}
```

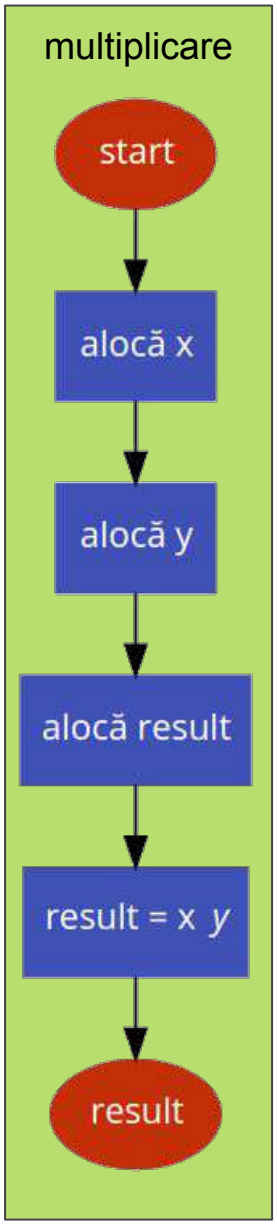



```
3 la puterea 5 este 243
```

Viața unui program

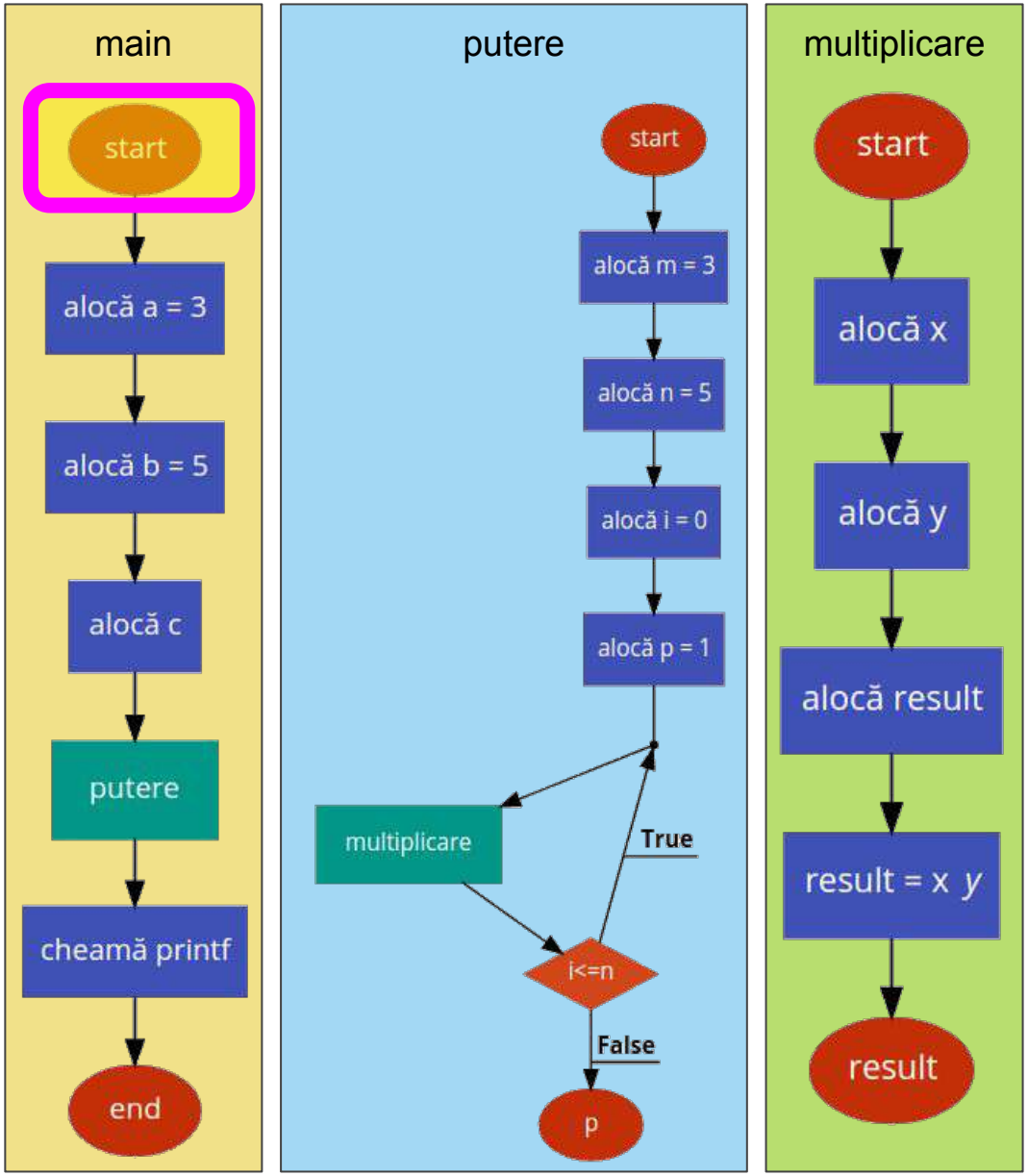


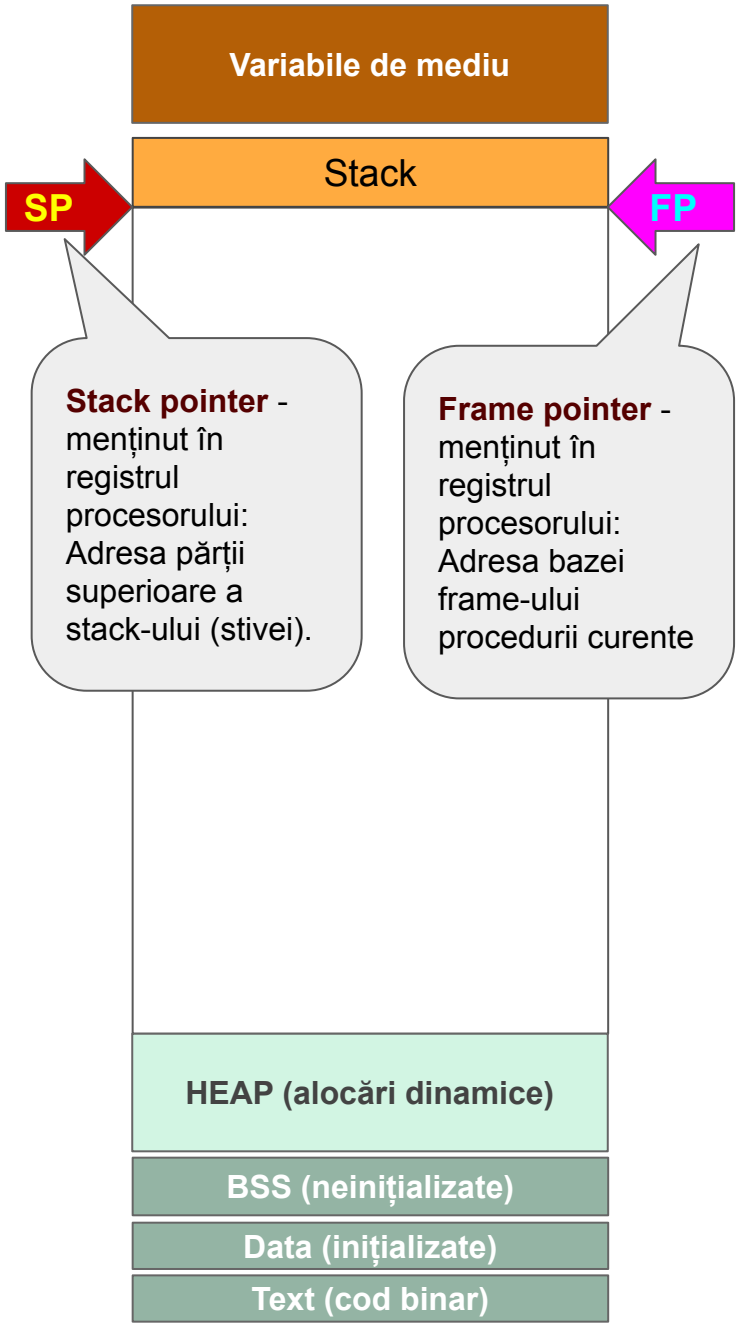




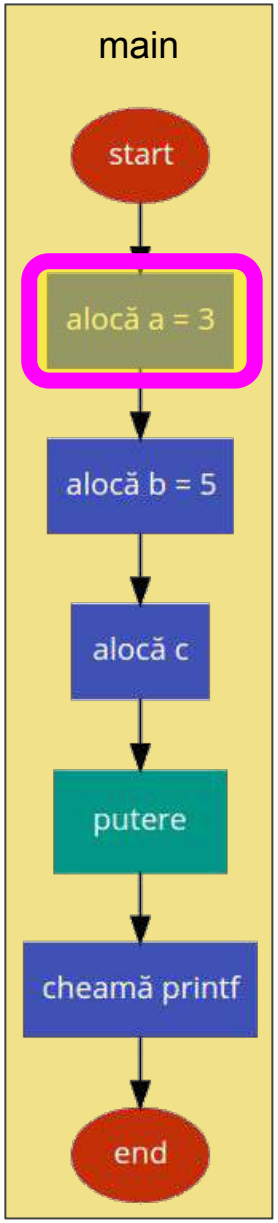

```
#include <stdio.h>

int multiplicare(int m, int n) {
   int result = 0;
   result = m * n;
   return result;
}

int putere(int m, int n)
{
  int i;
  int p=1;
  for (i=1; i<=n; i++) {
      p = multiplicare(p,m);
  }
  return p;
}

int main(int argc, char *argv[])
{
  int a=3;
  int b=5;
  int c = putere(a, b);
  printf("%d la puterea %d este
%d\n",a,b,c);
  return 0;
}
```

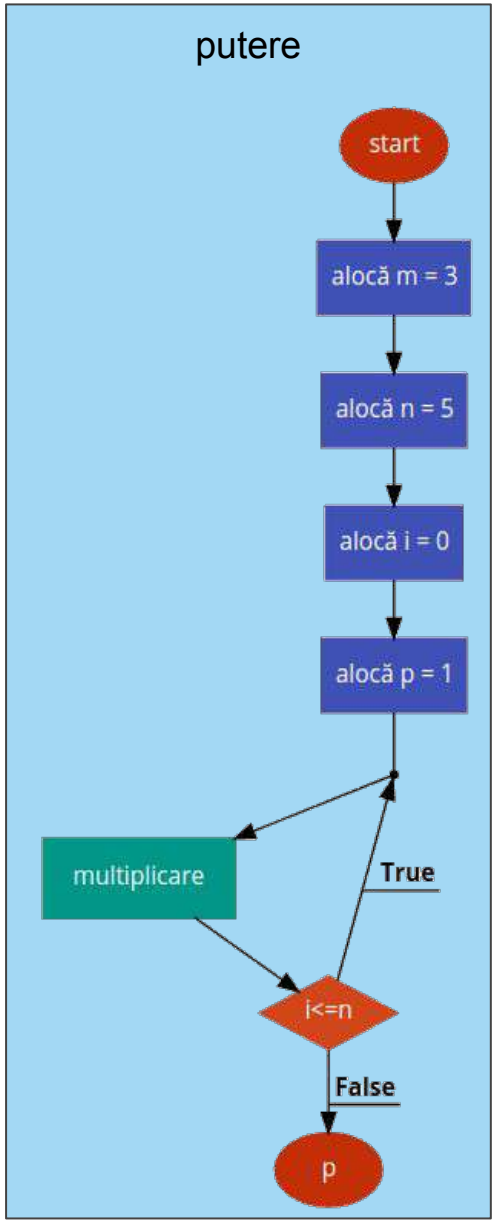
Viața unui program

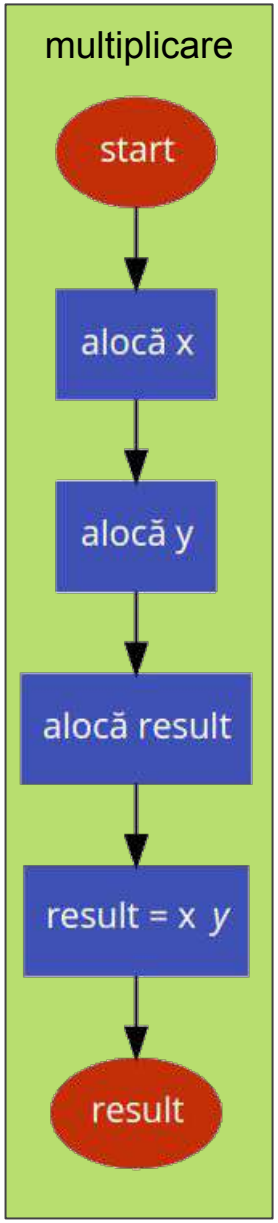


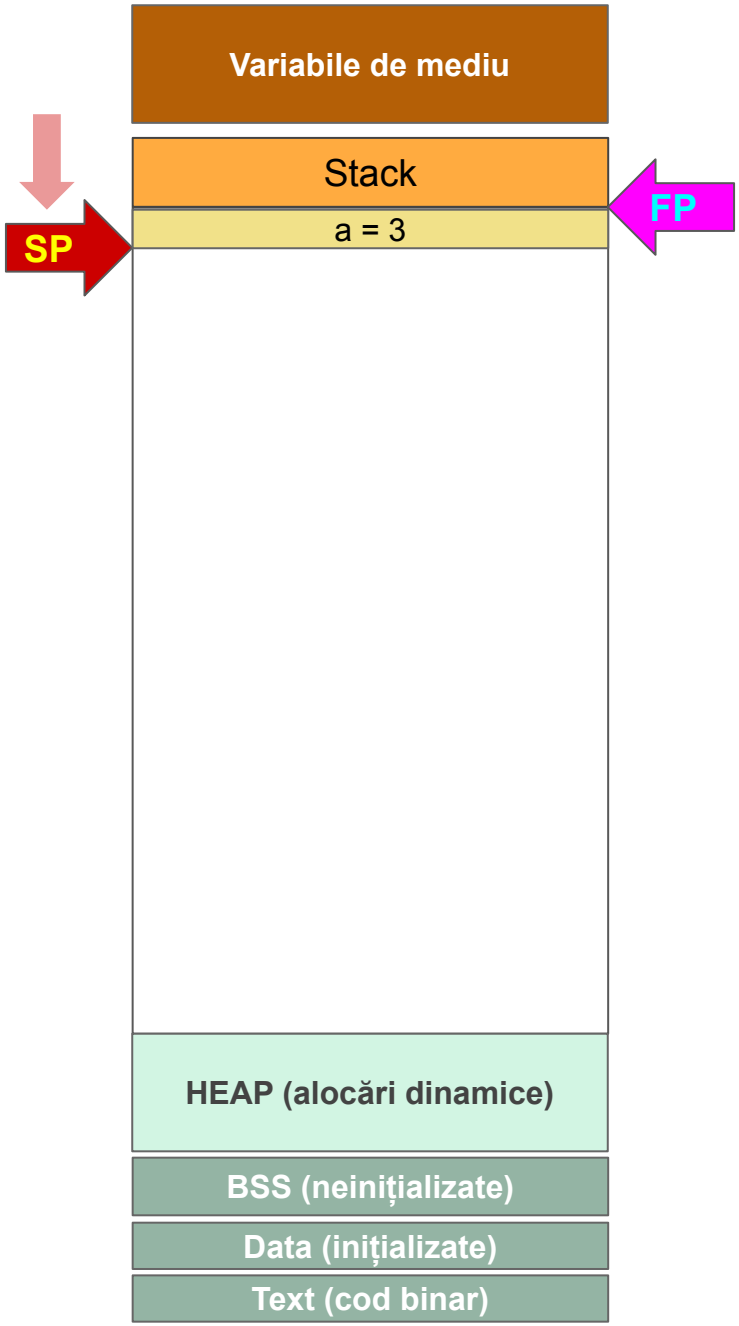


Viața unui program

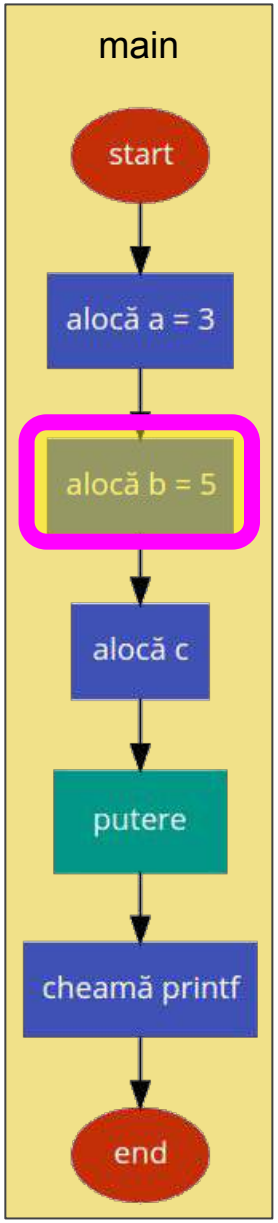


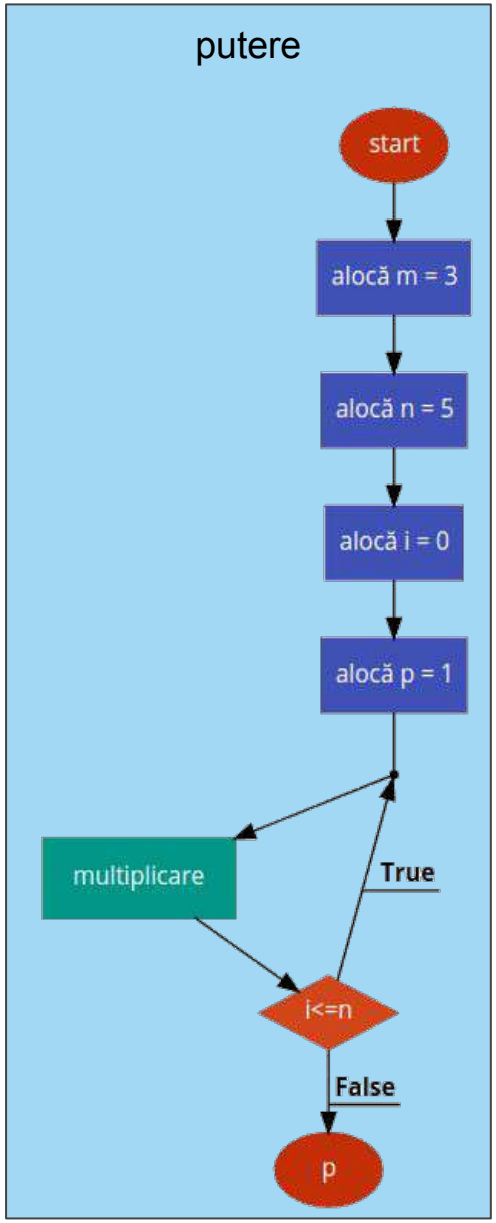


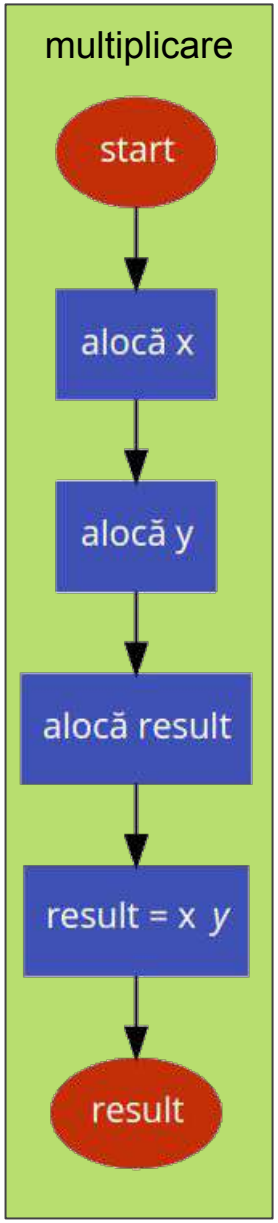


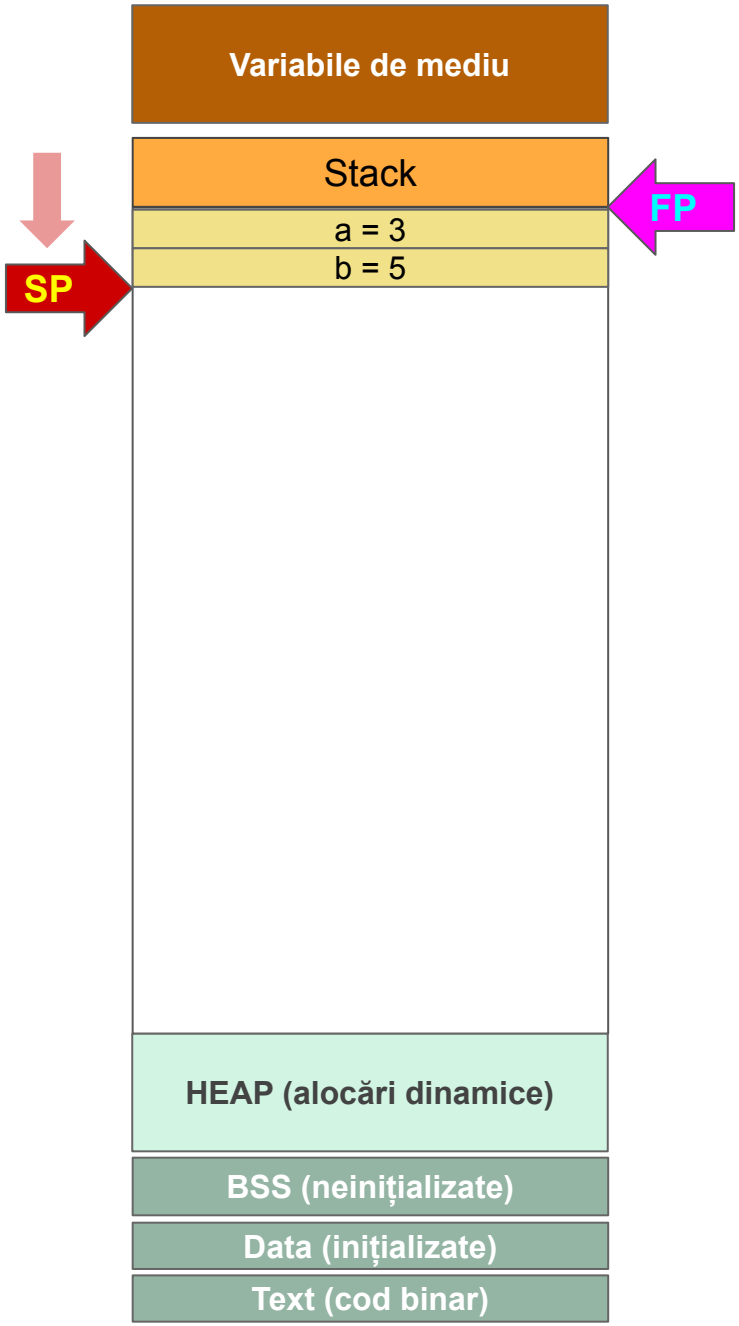


Viața unui program

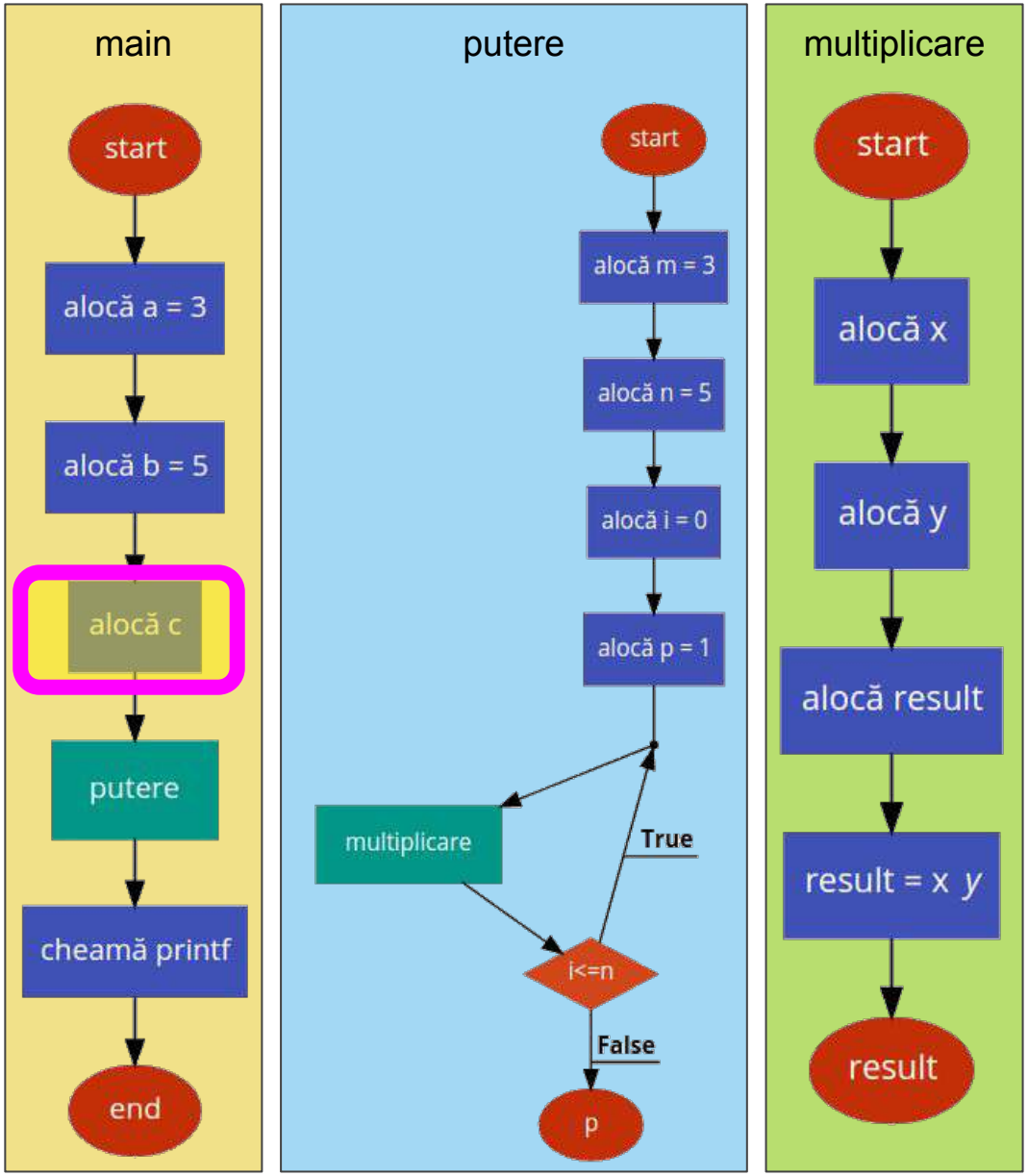


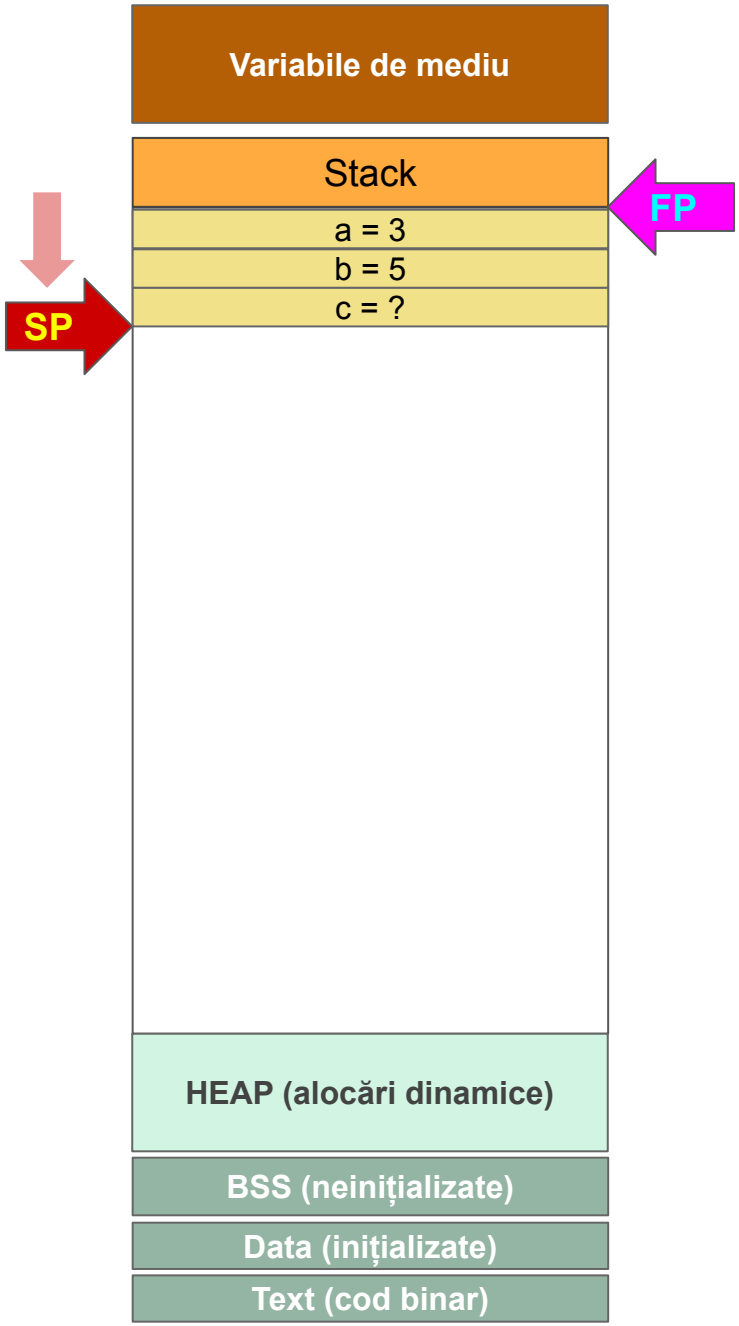




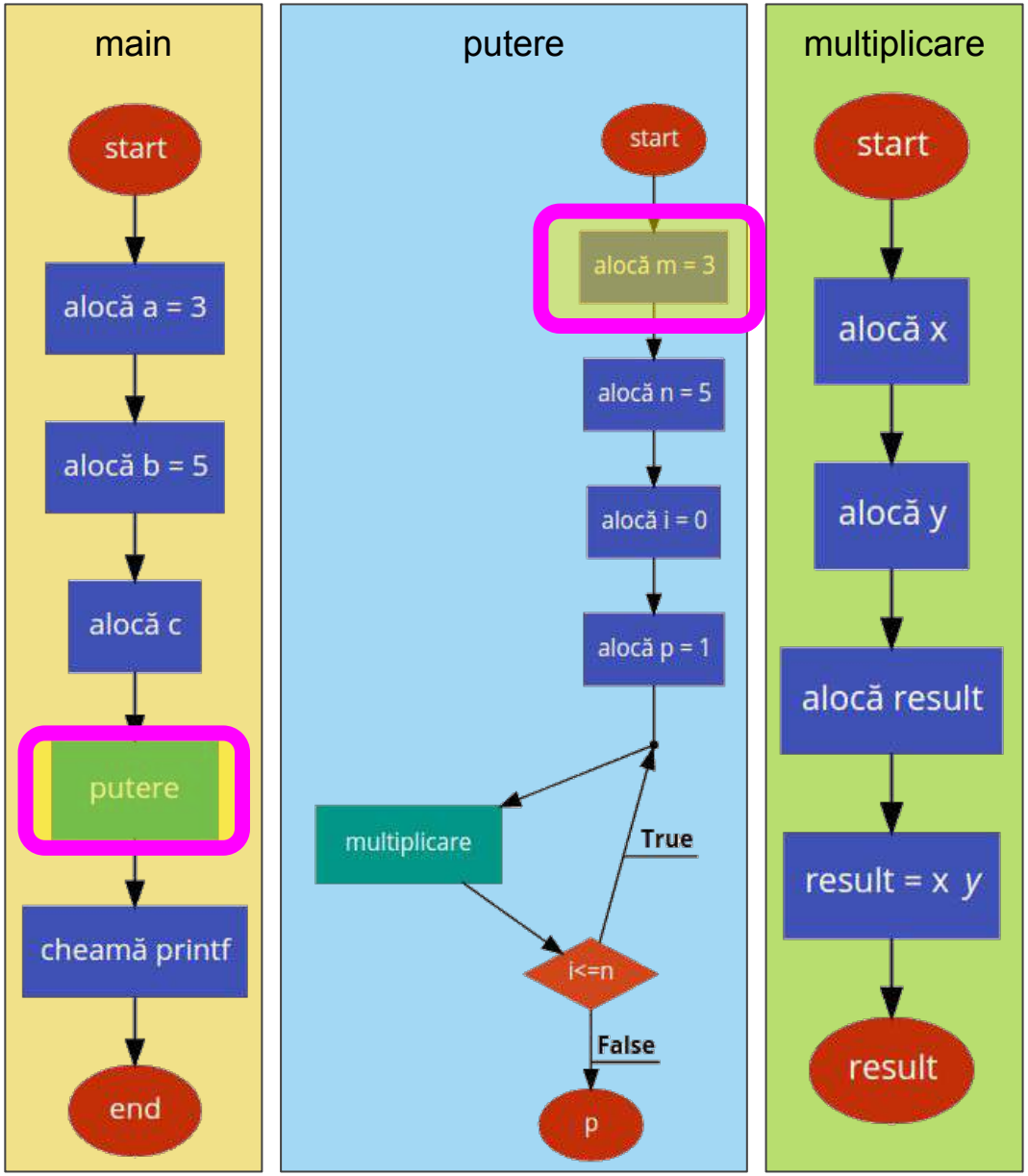


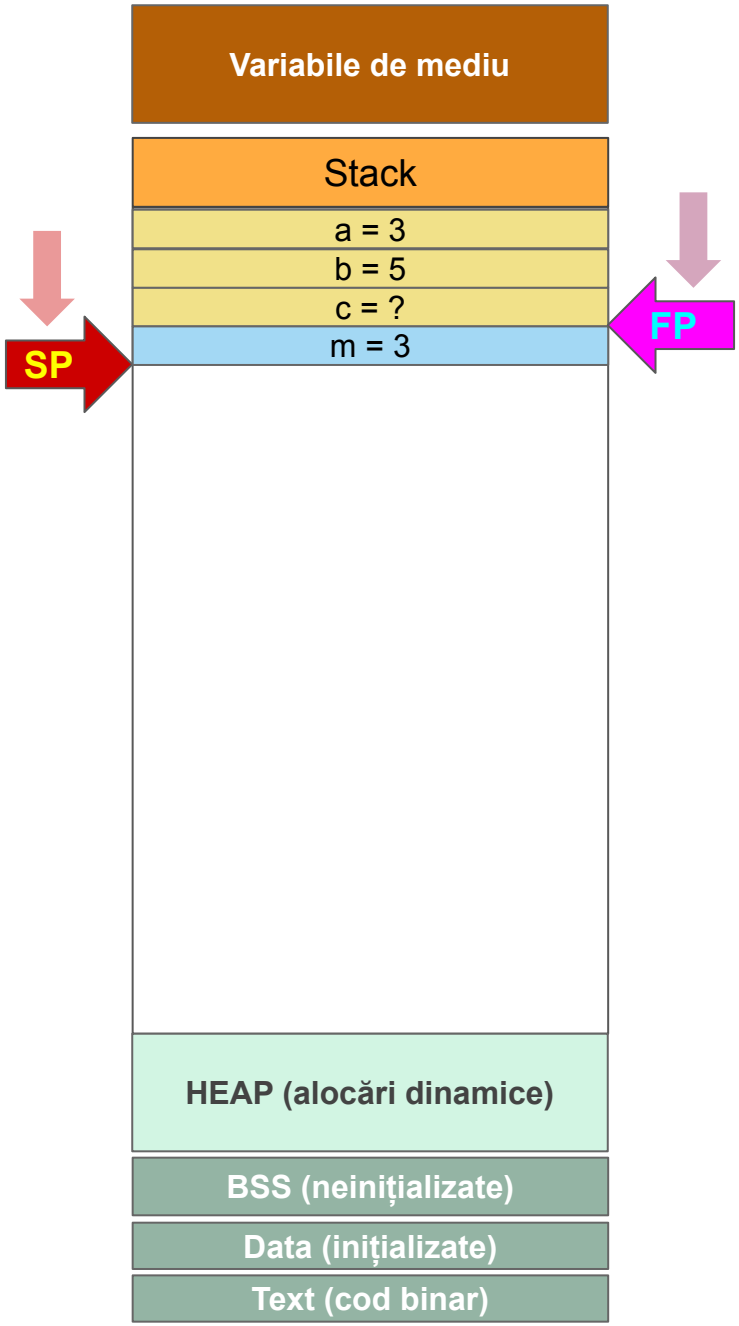
Viața unui program



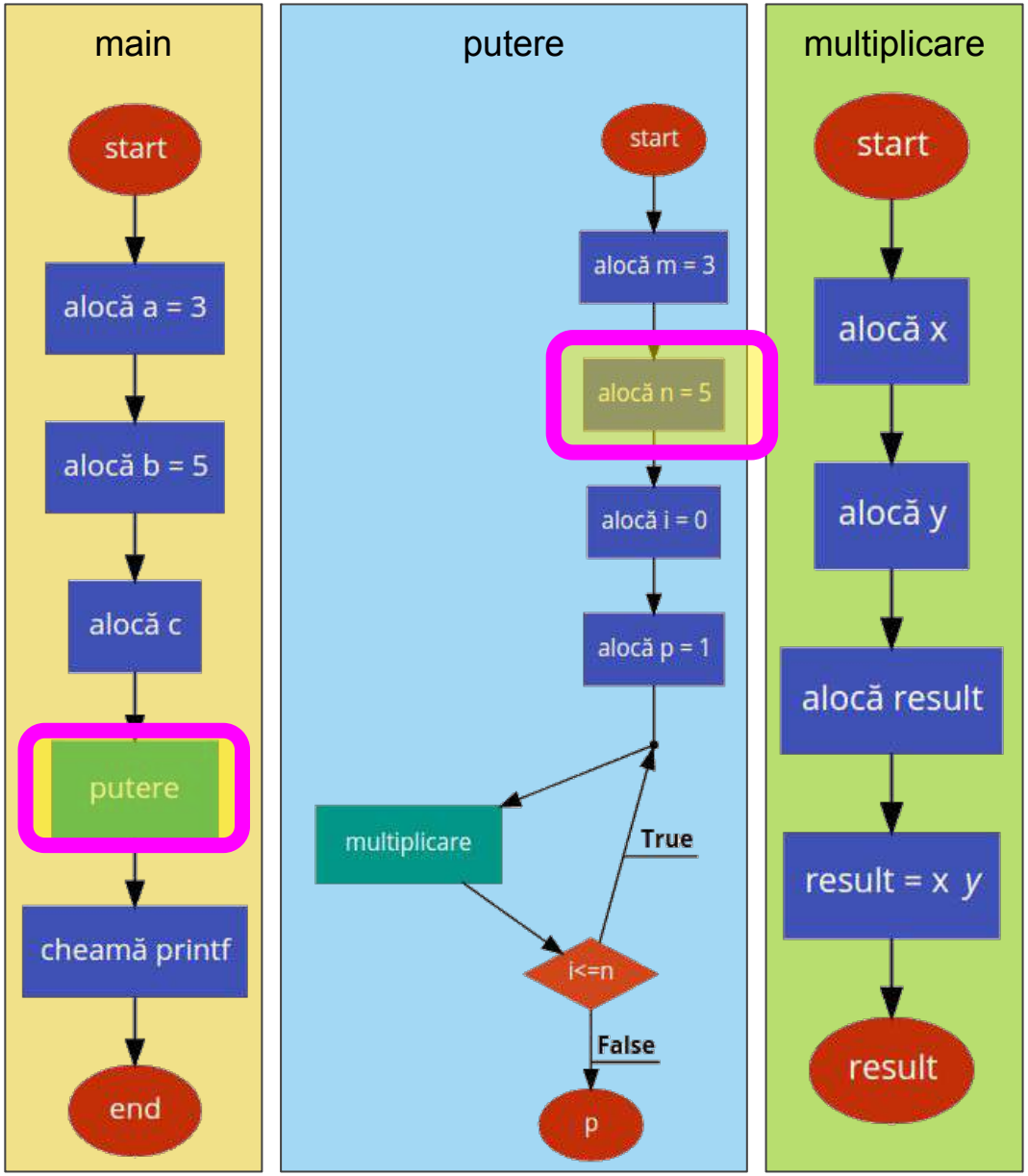


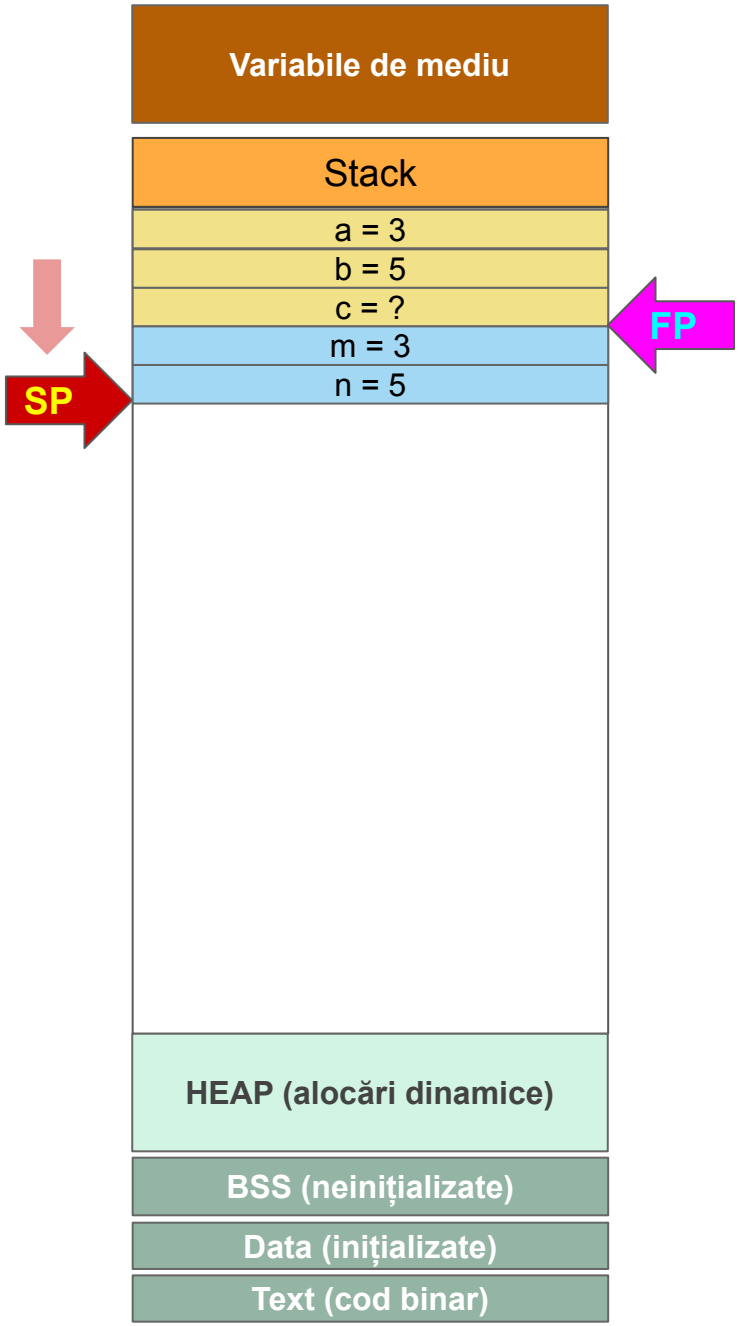
Viața unui program



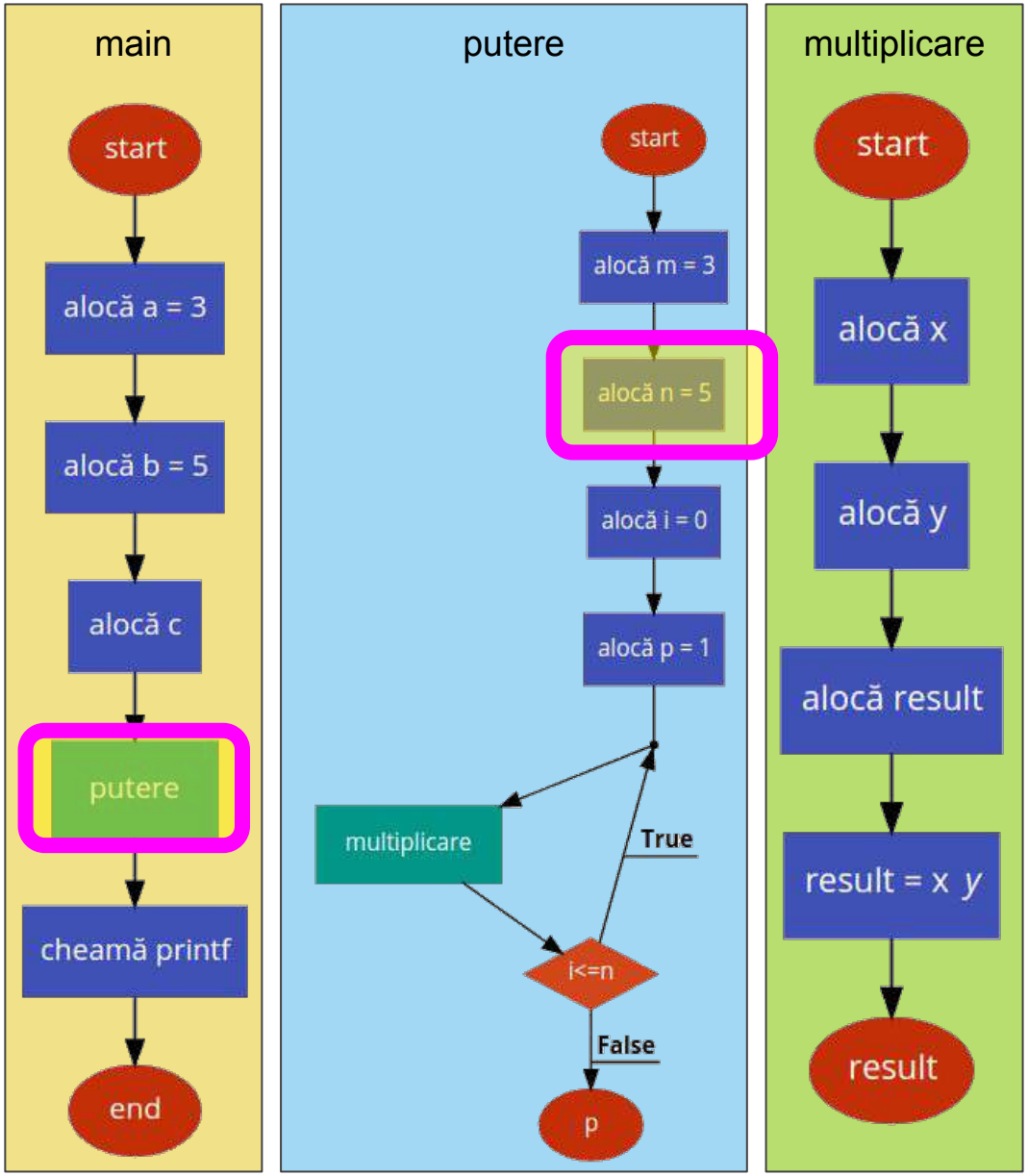


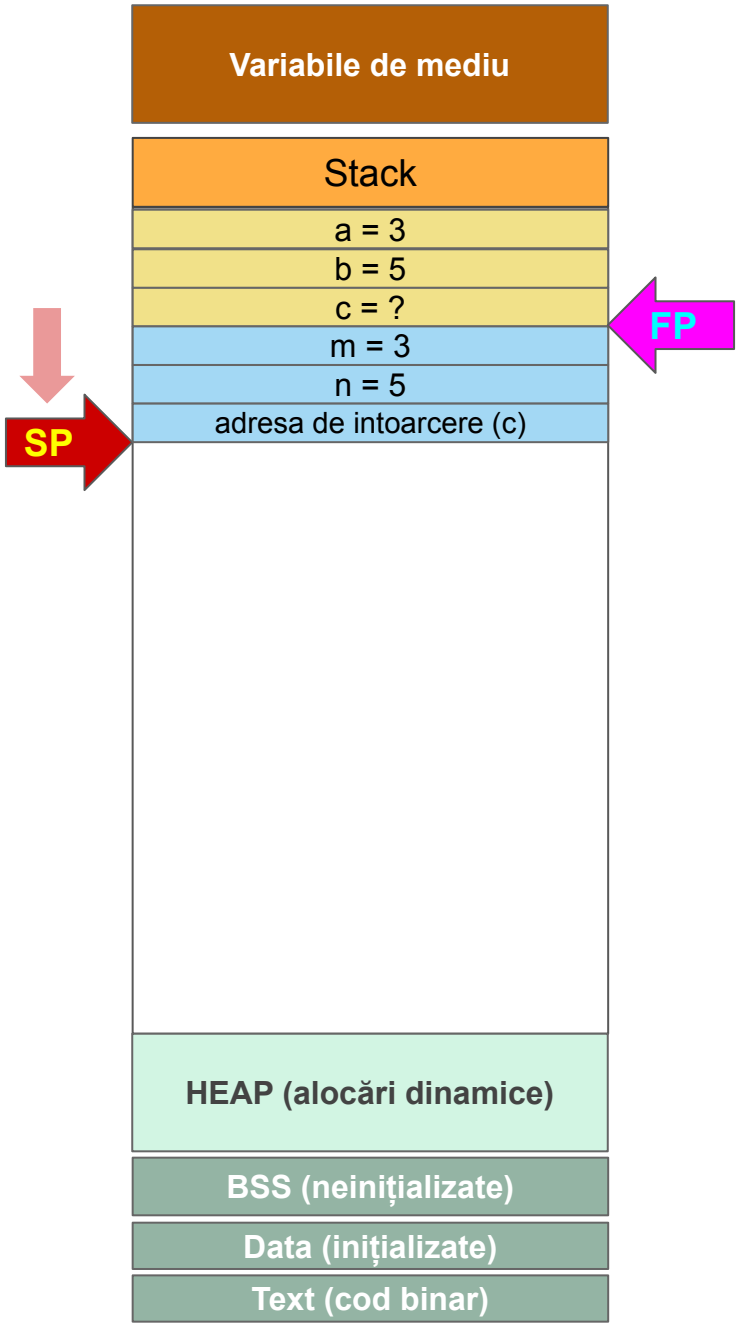
Viața unui program



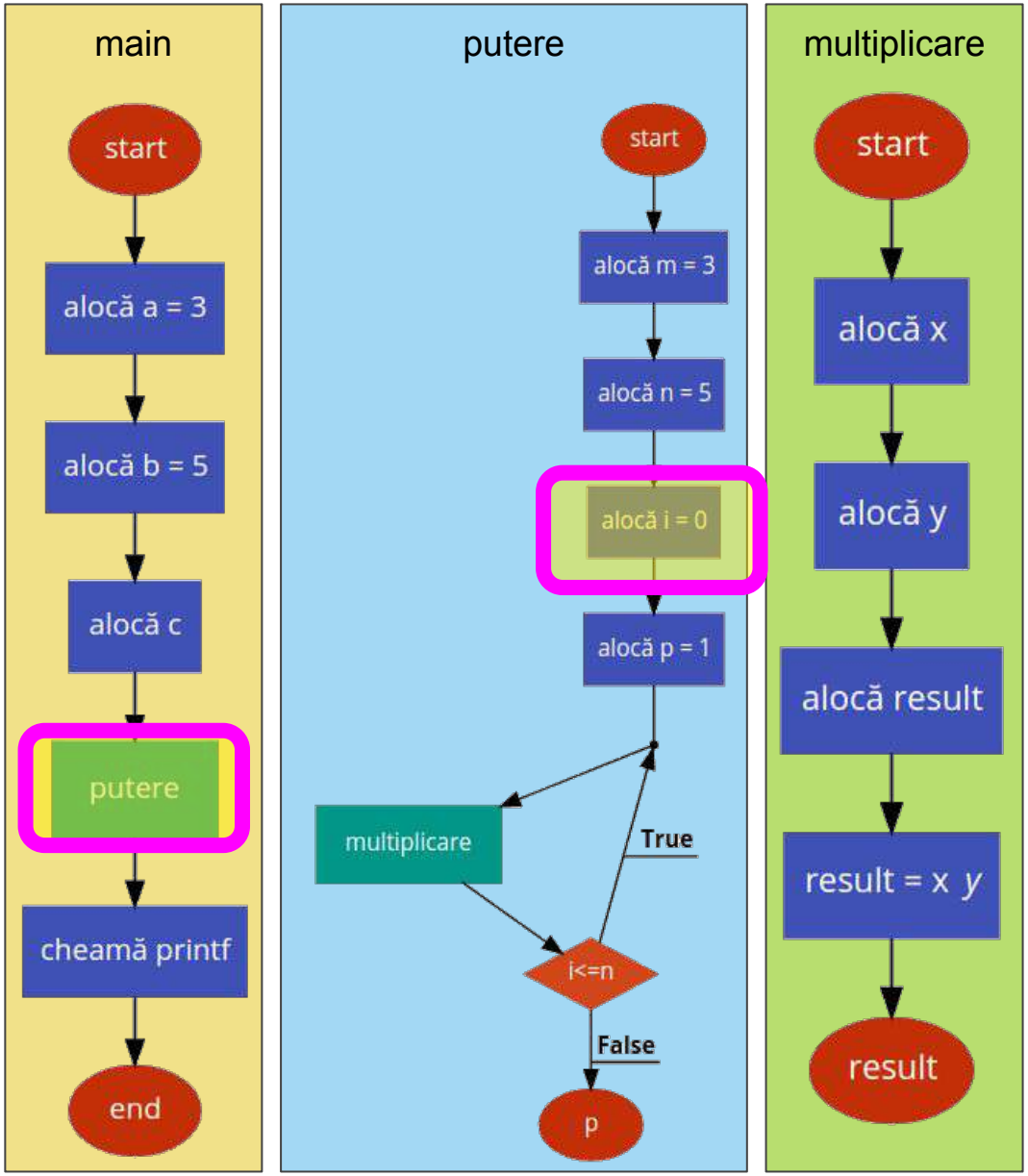


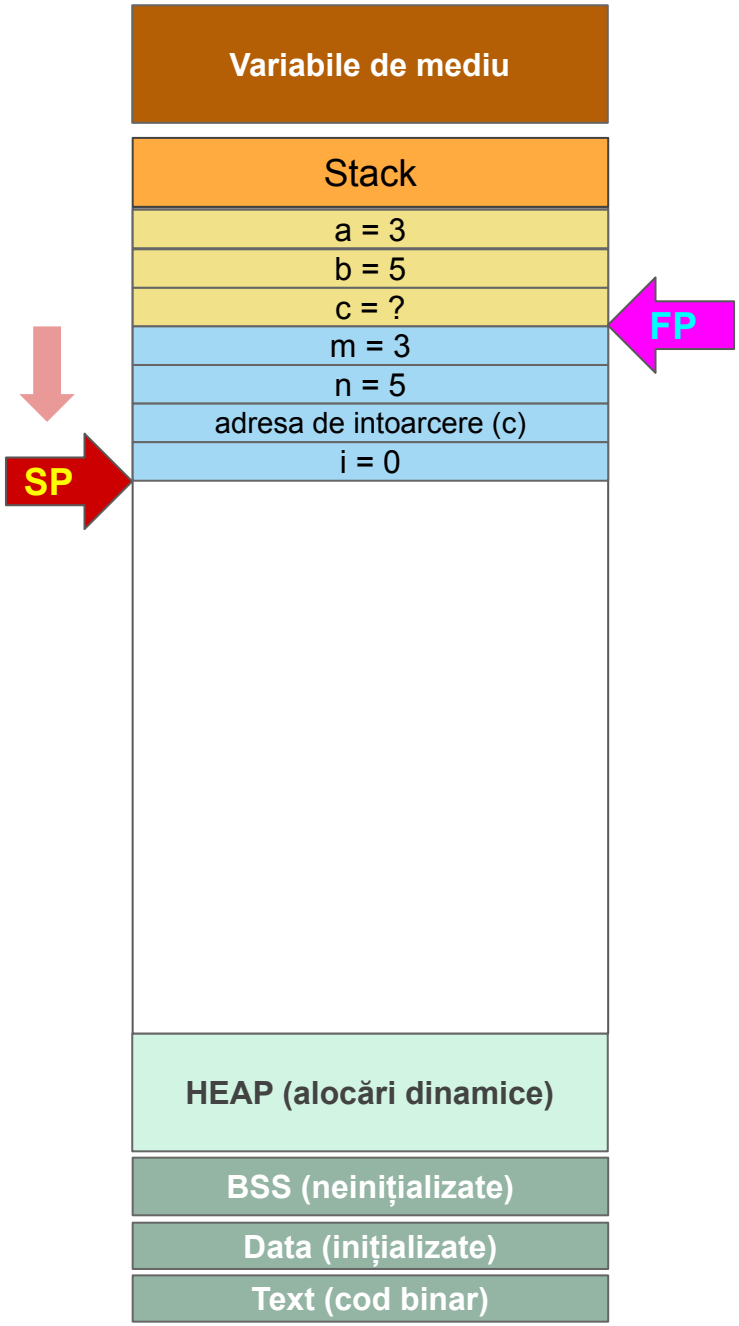
Viața unui program



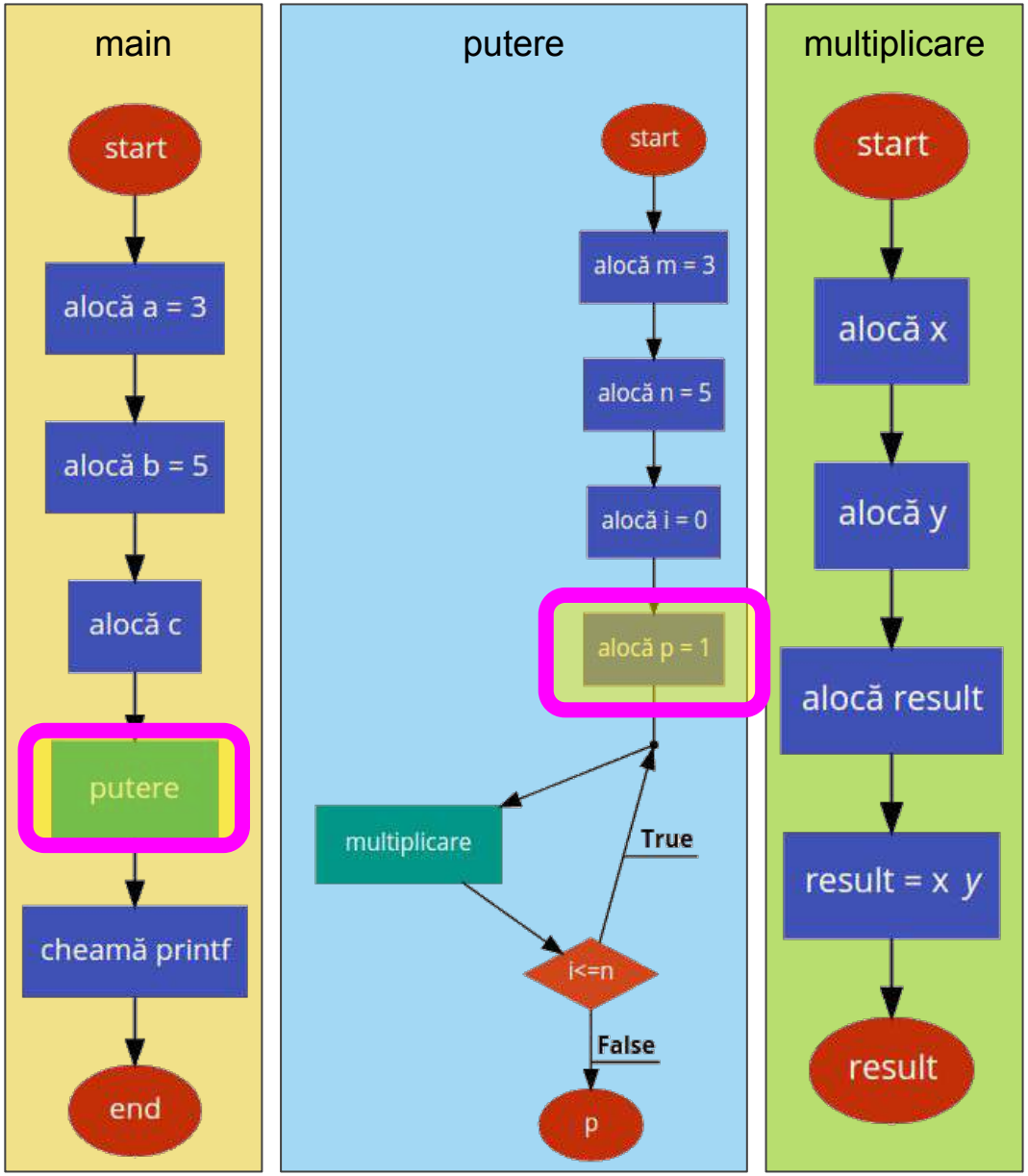


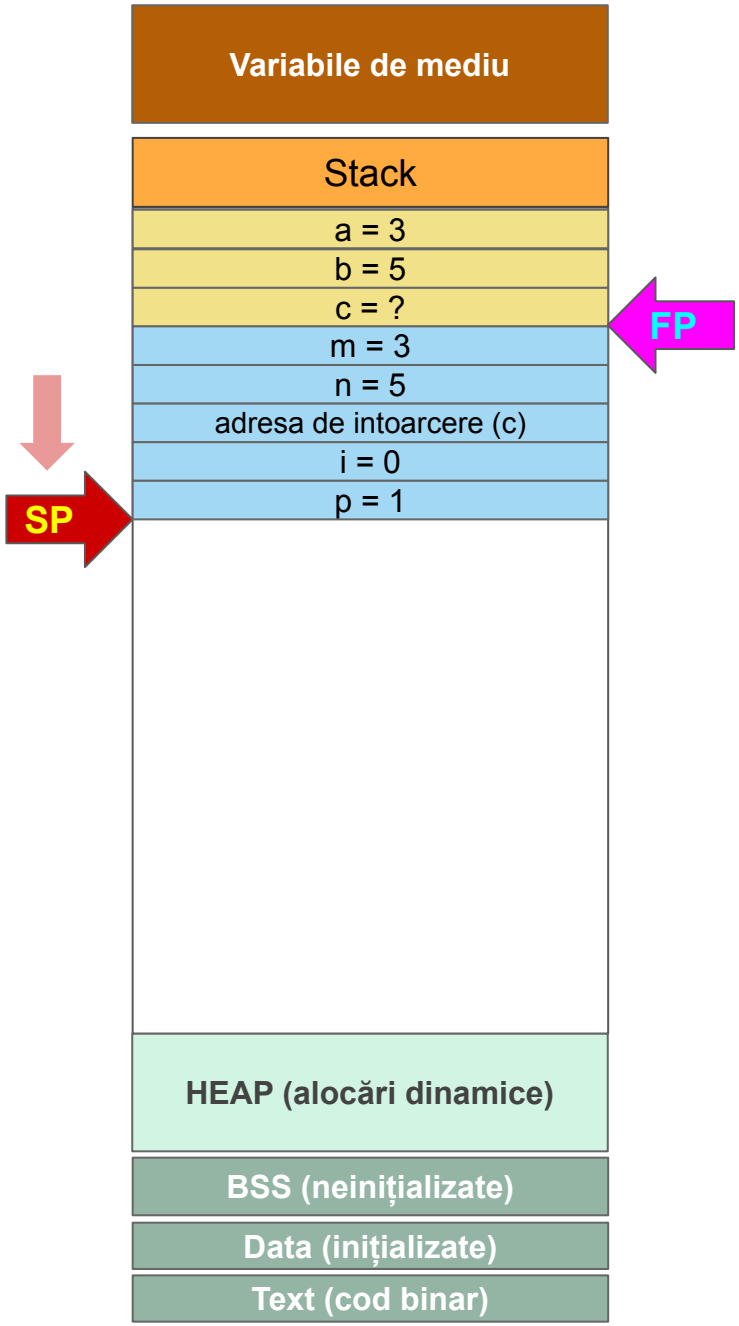
Viața unui program



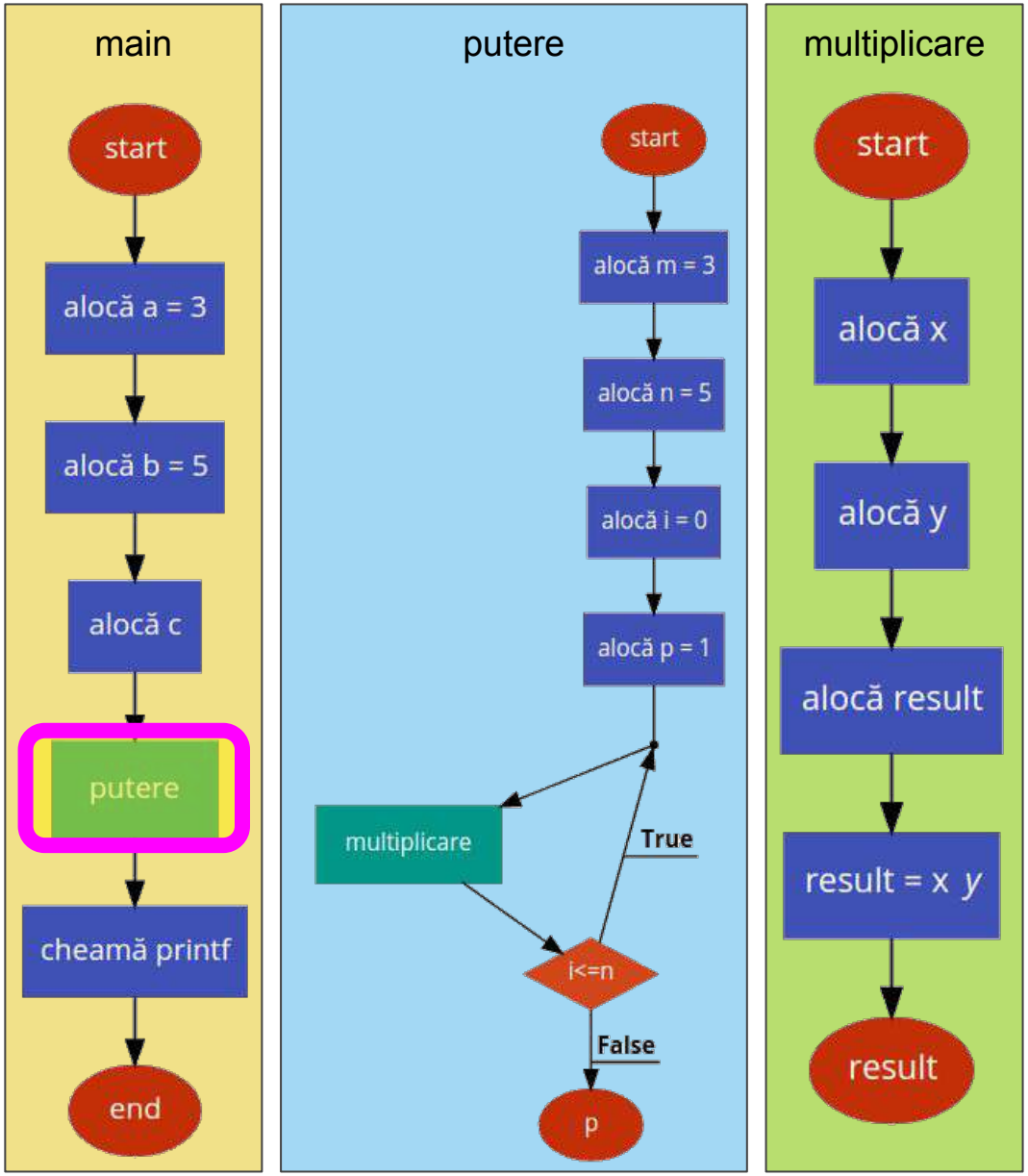


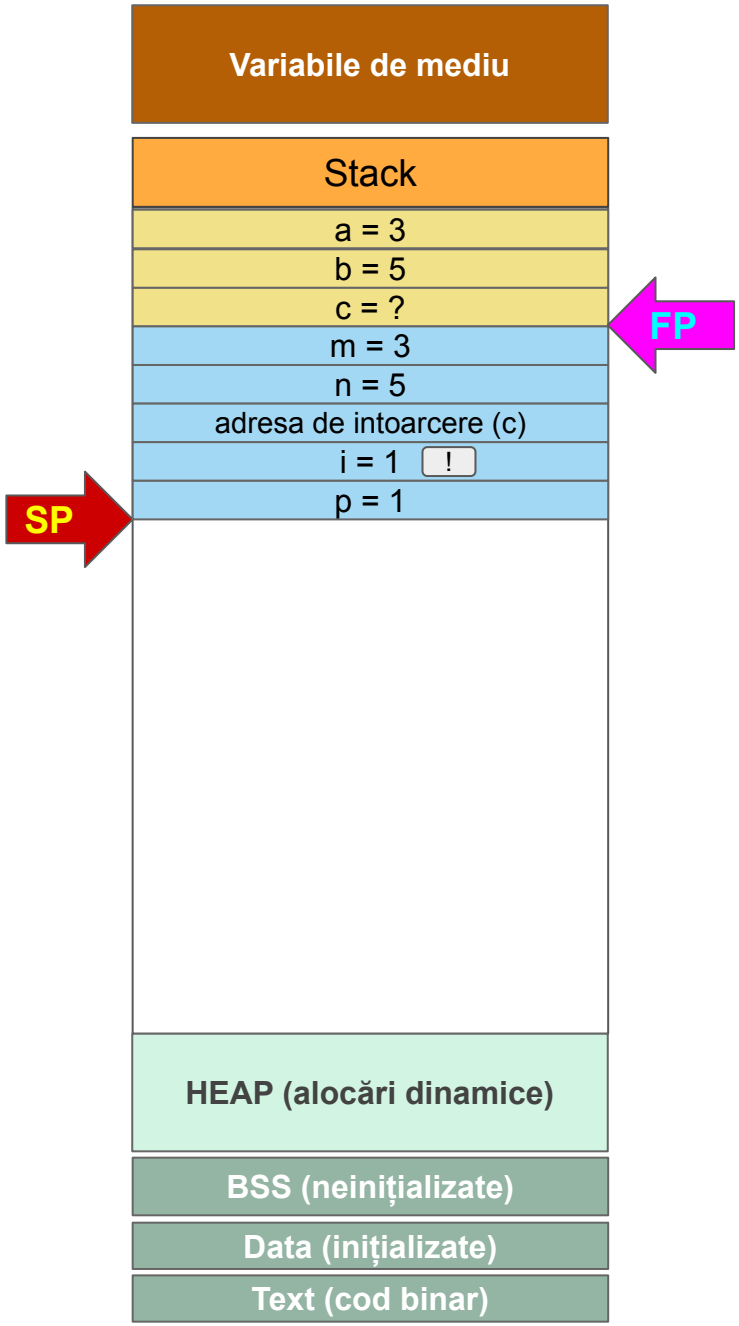
Viața unui program



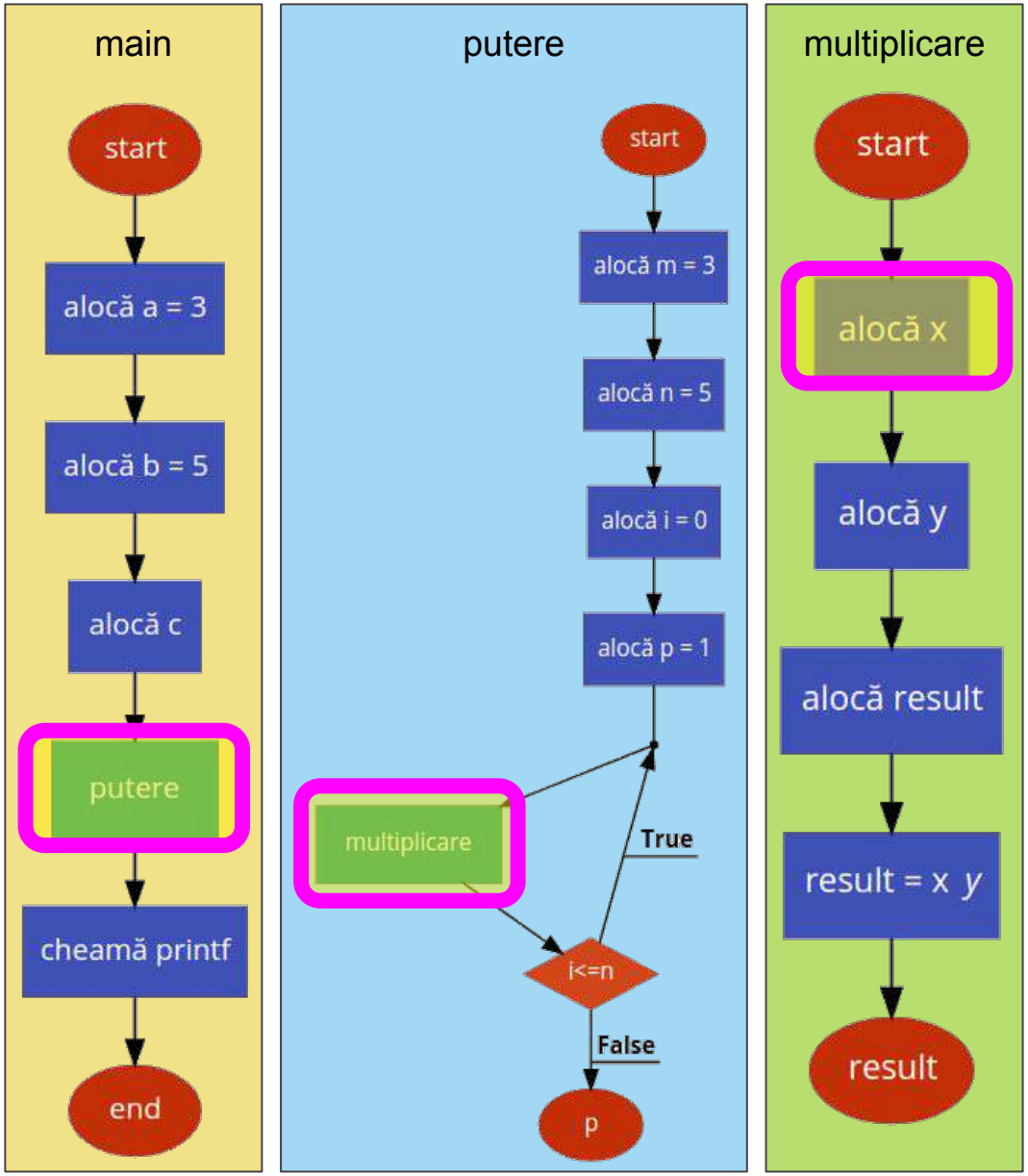


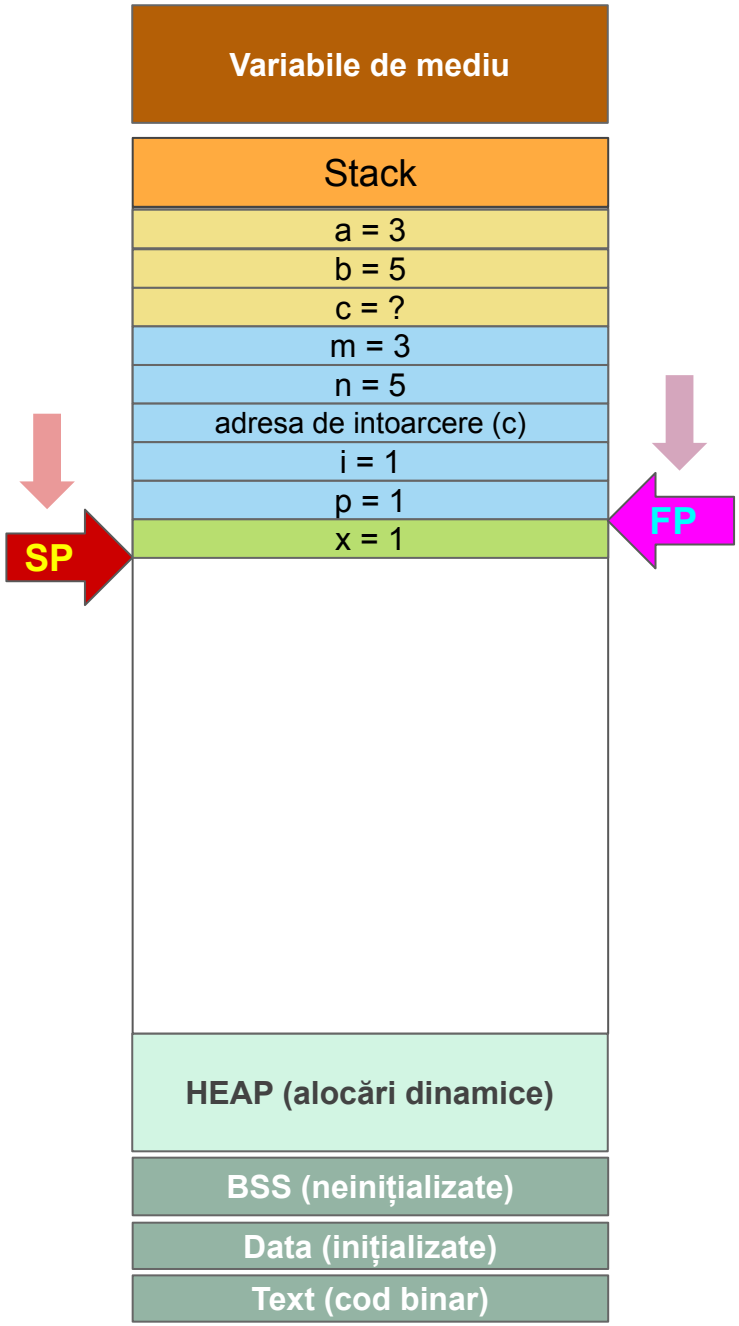
Viața unui program



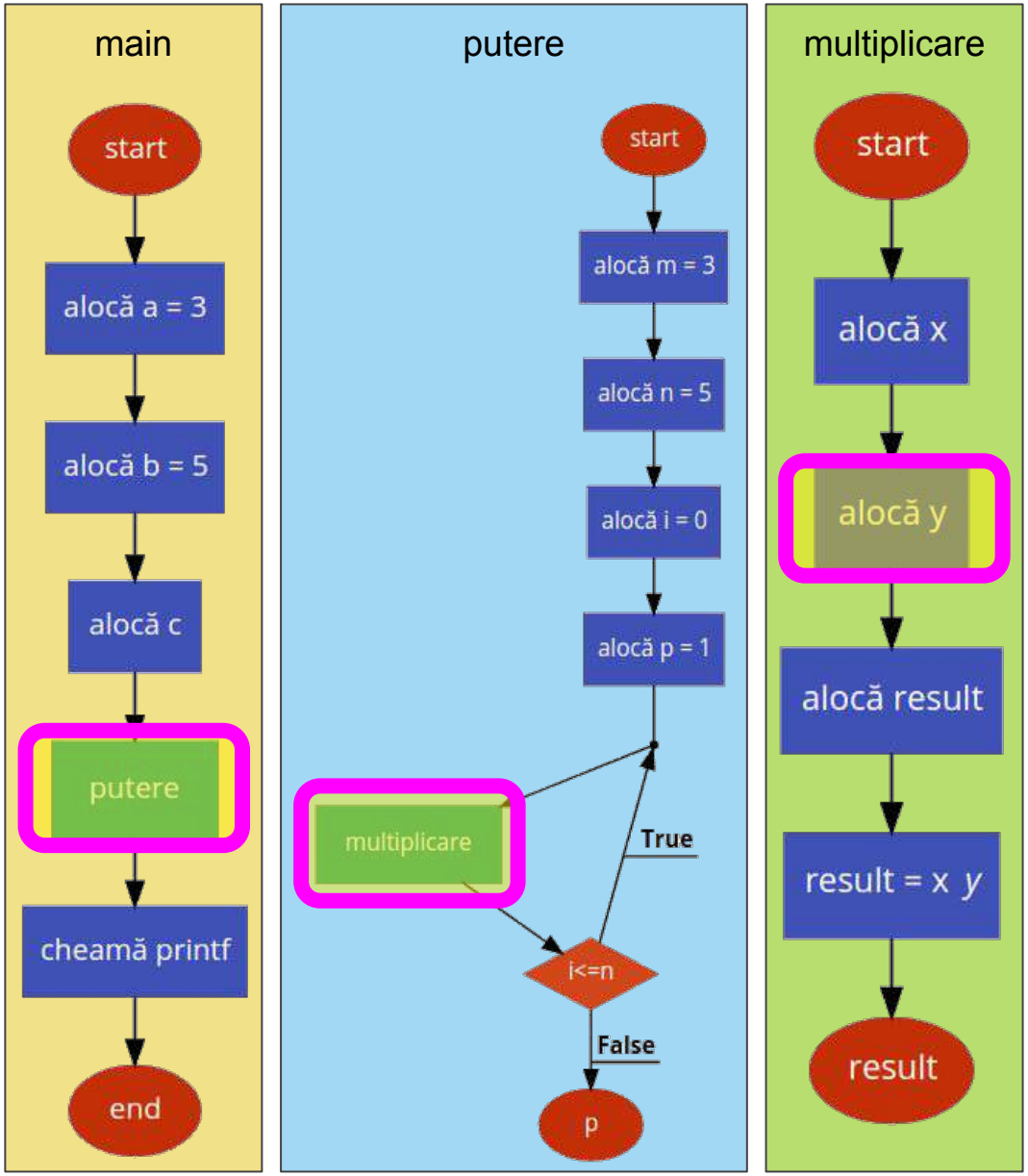


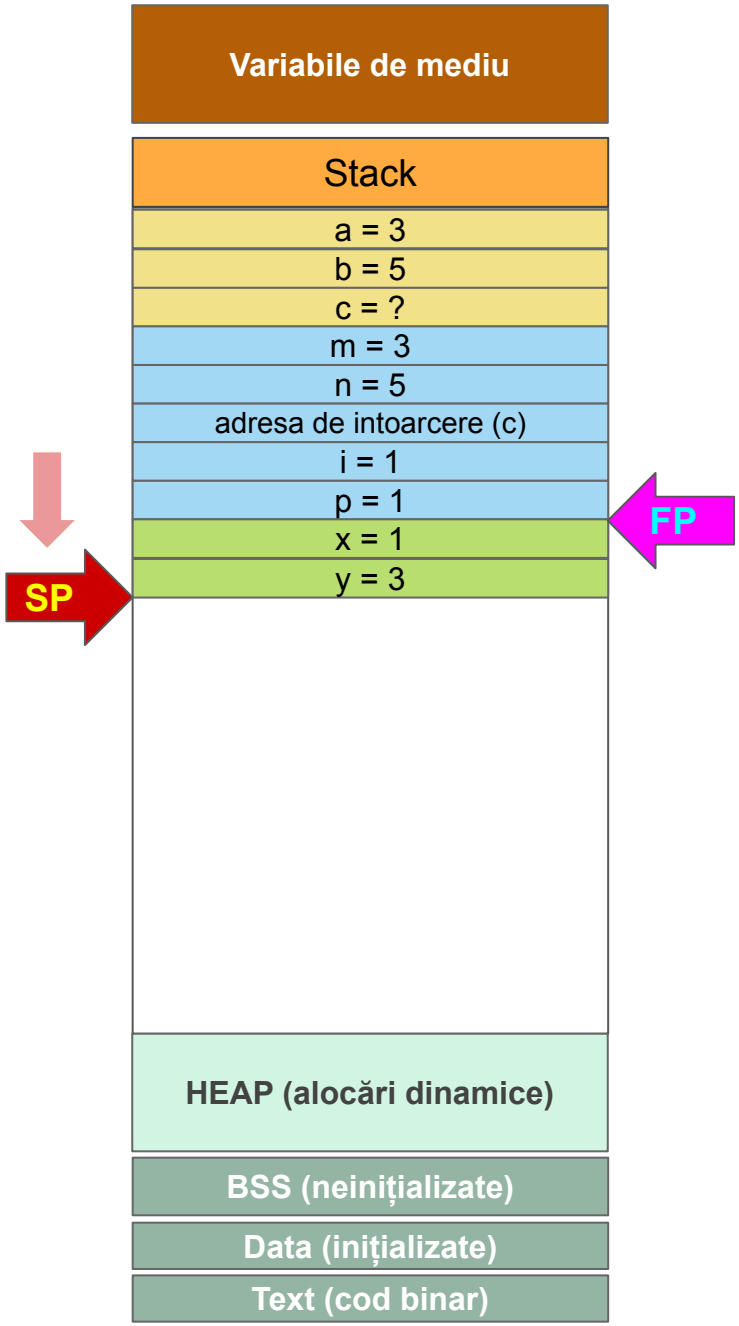
Viața unui program



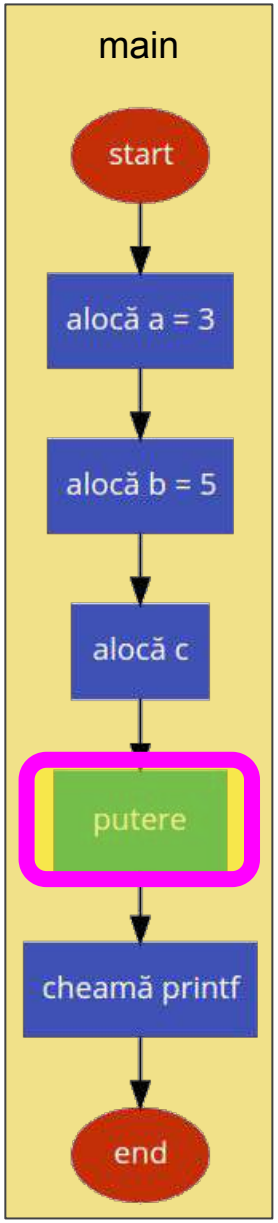


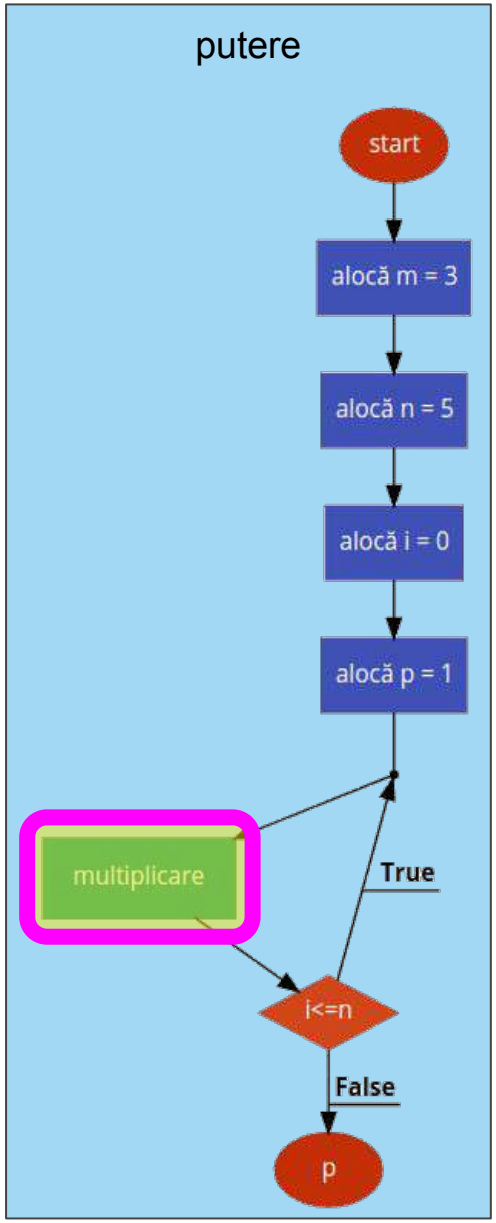
Viața unui program

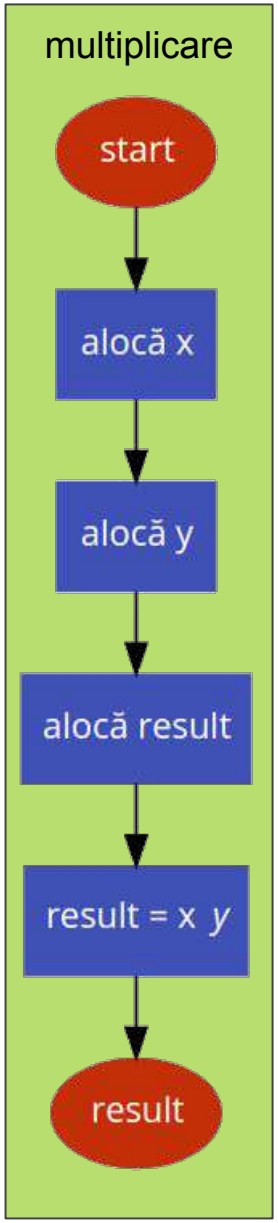


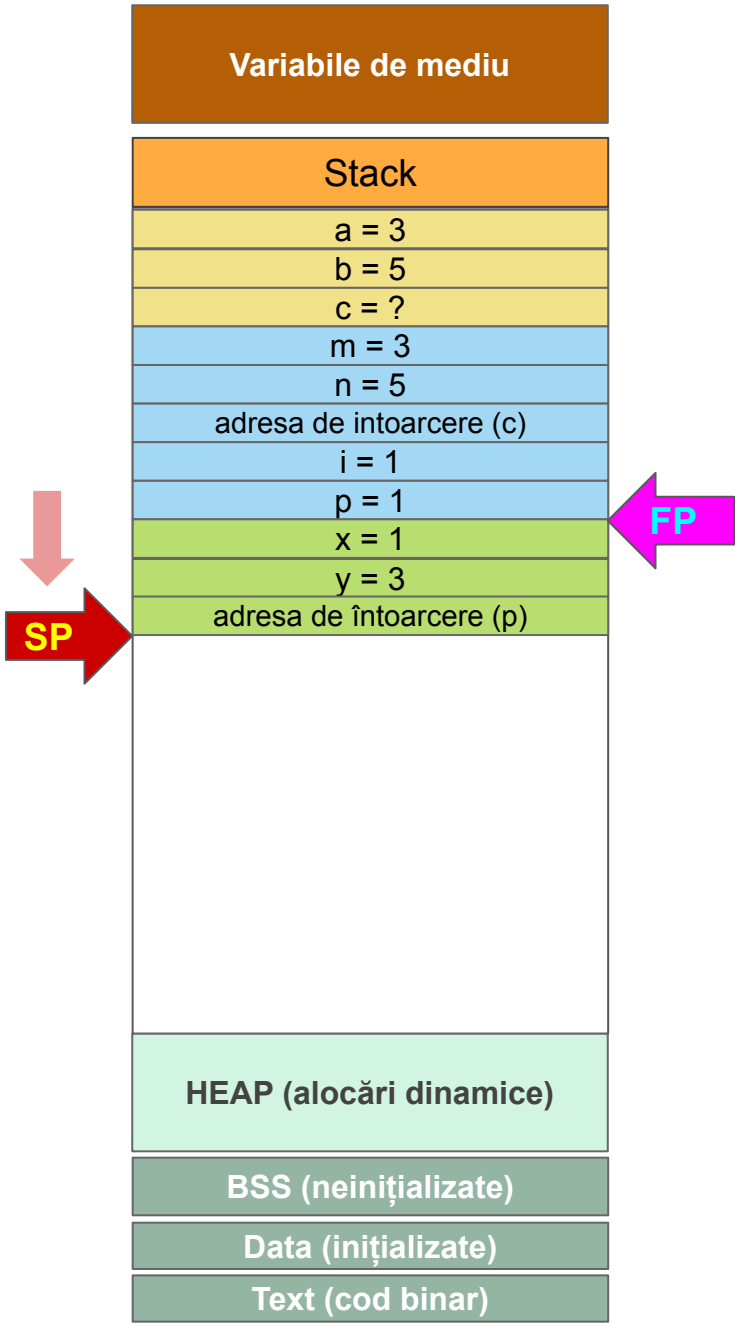


Viața unui program

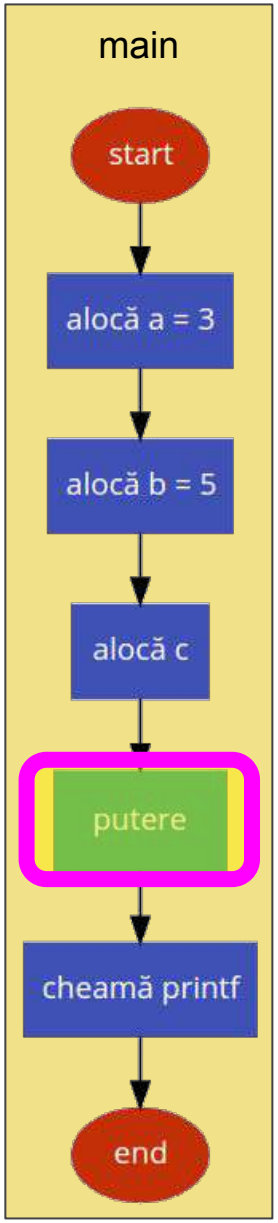


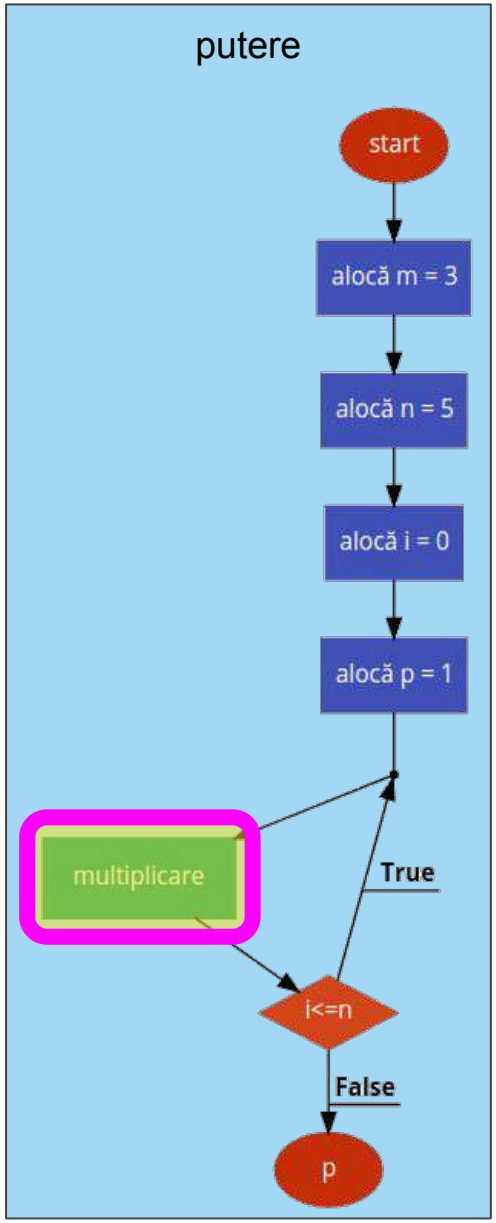


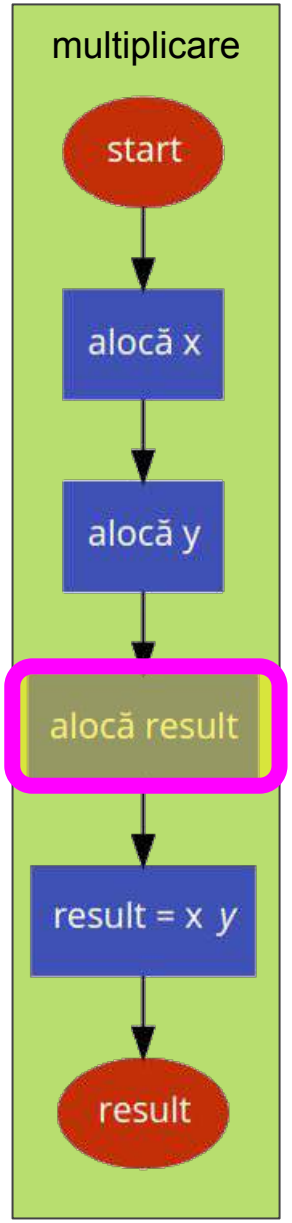


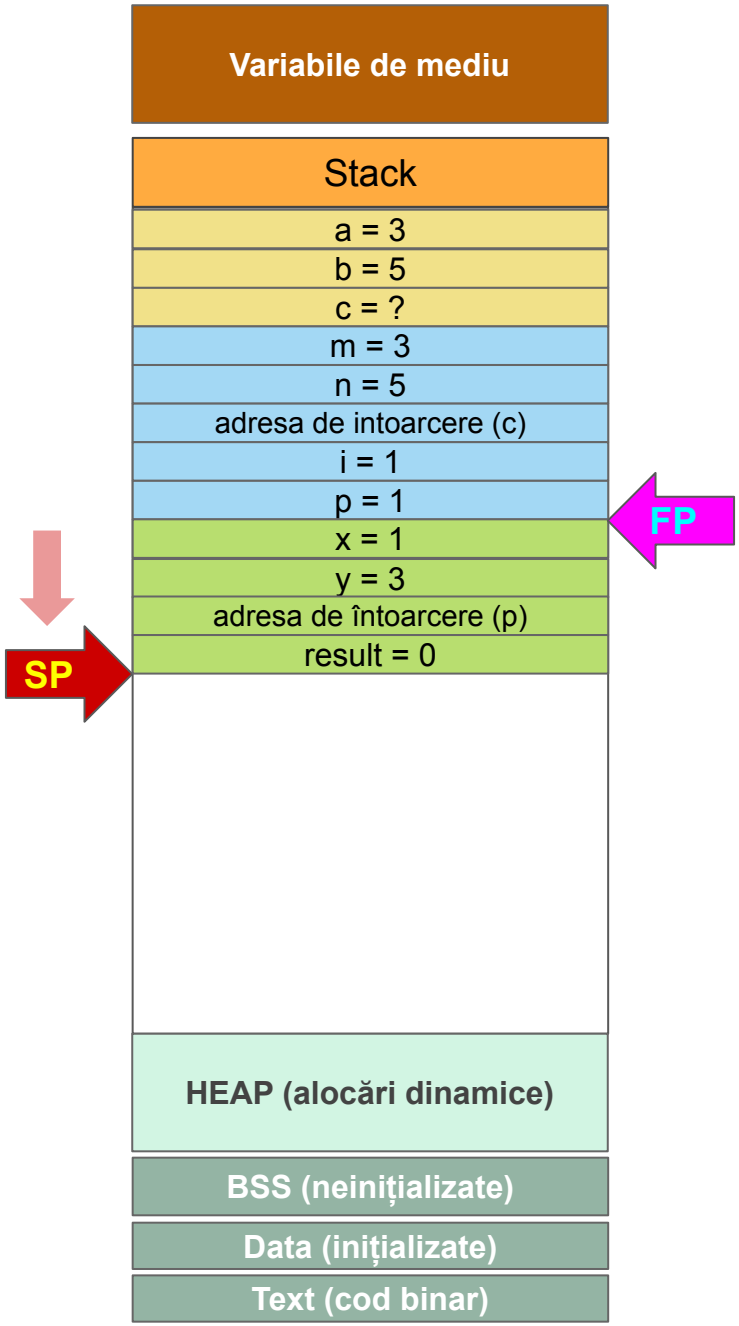


Viața unui program

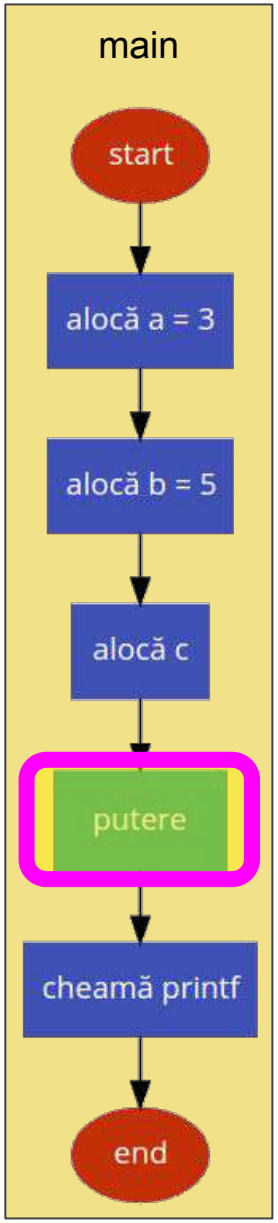


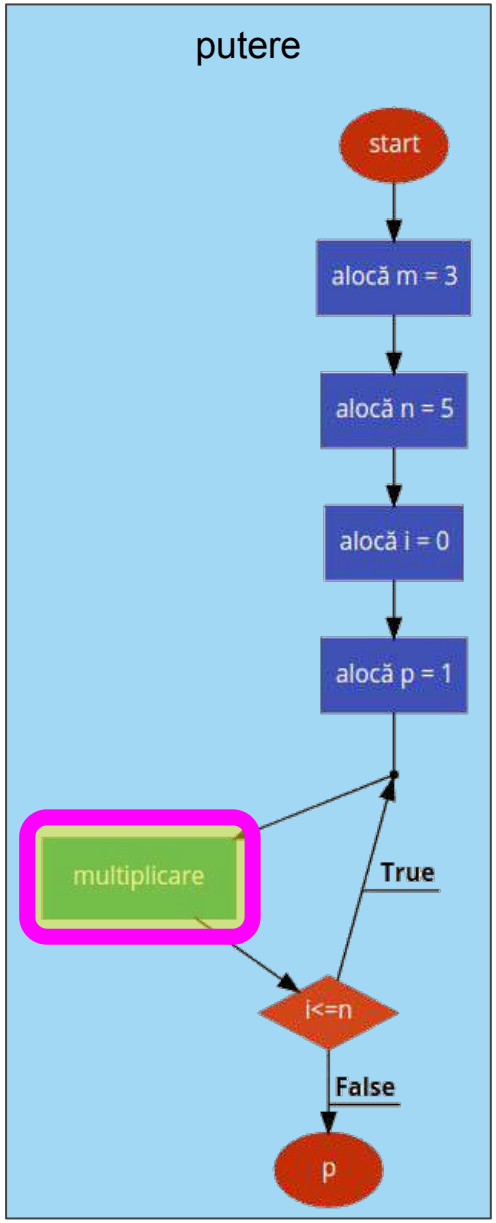


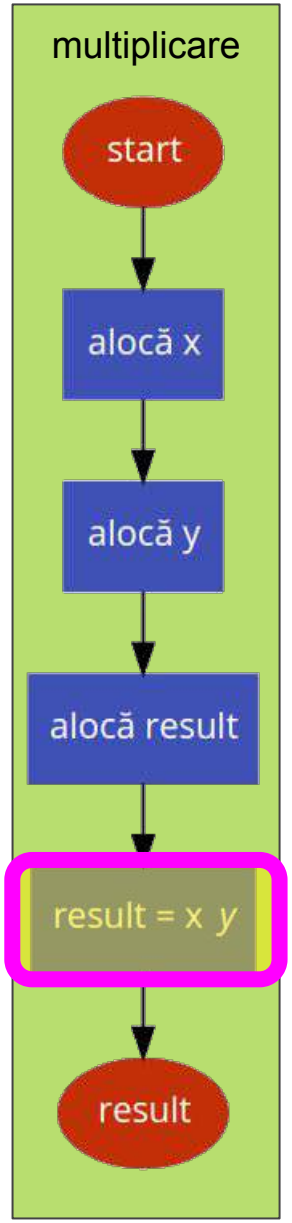


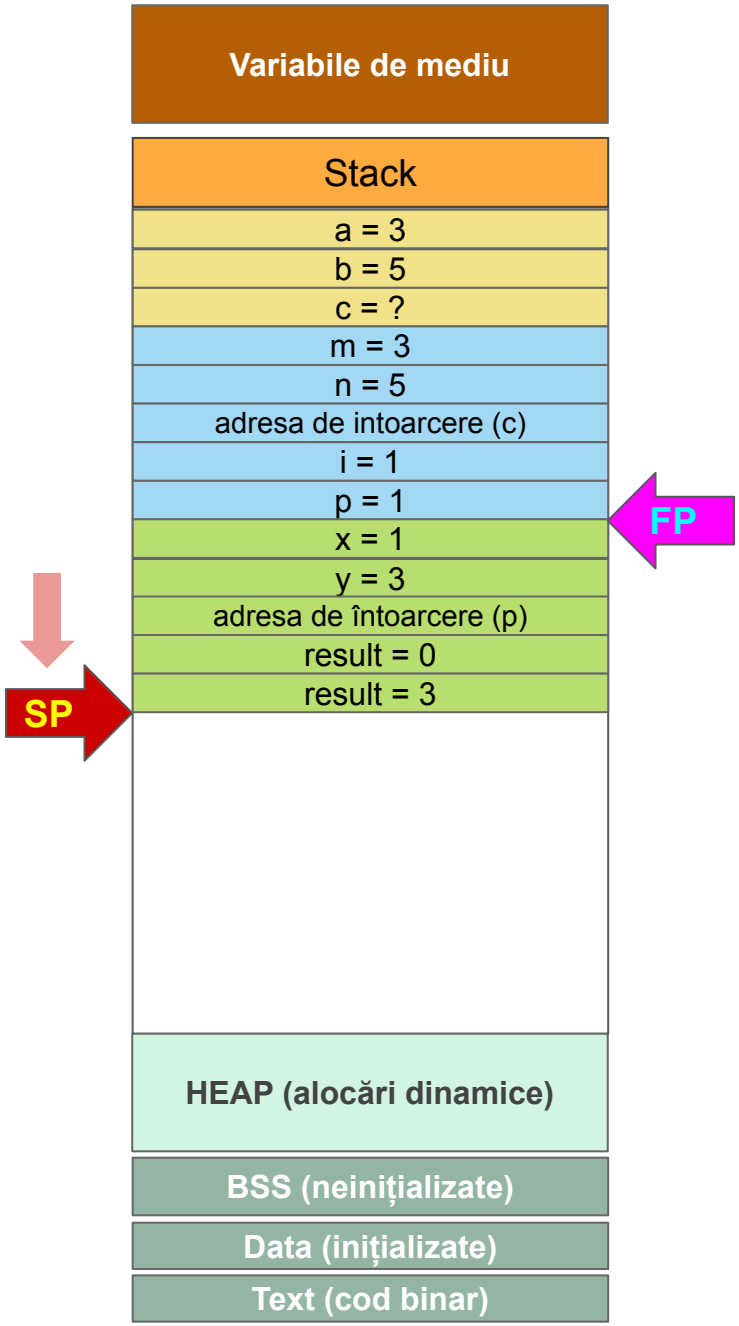


Viața unui program

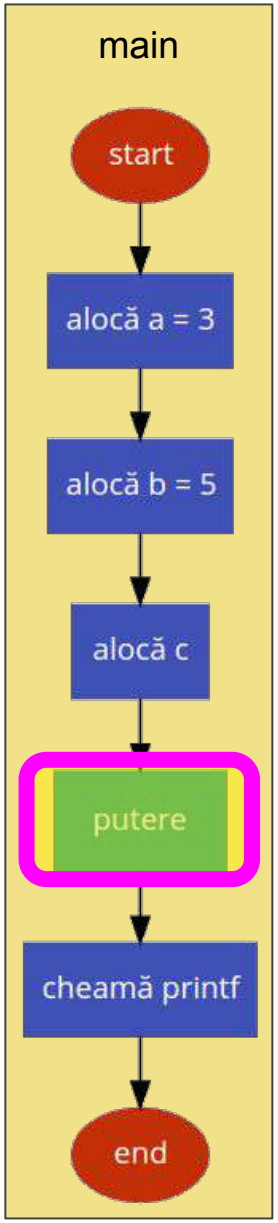


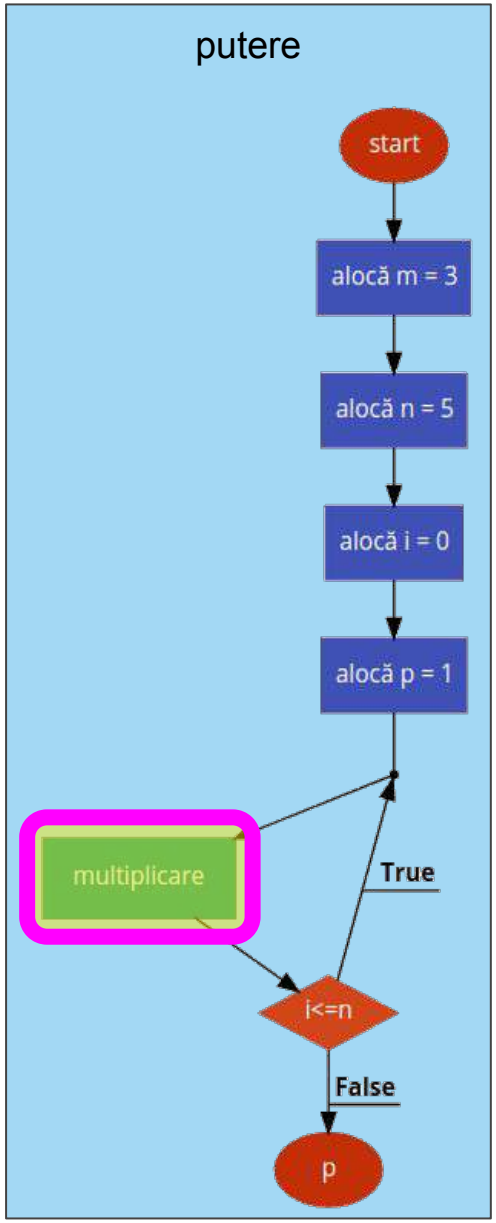


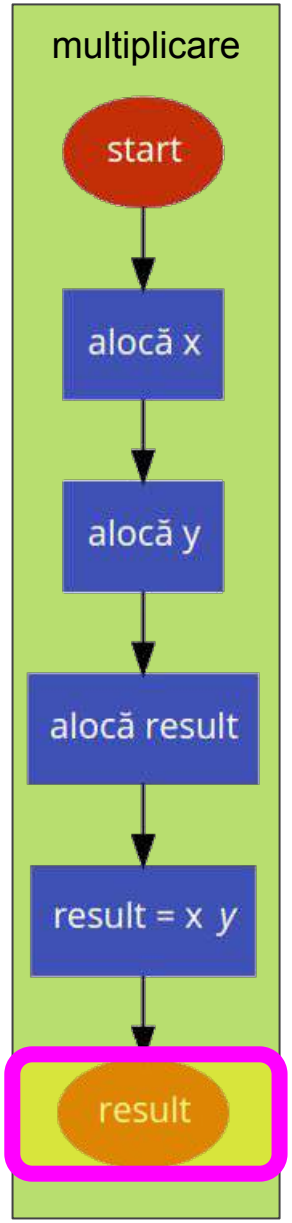


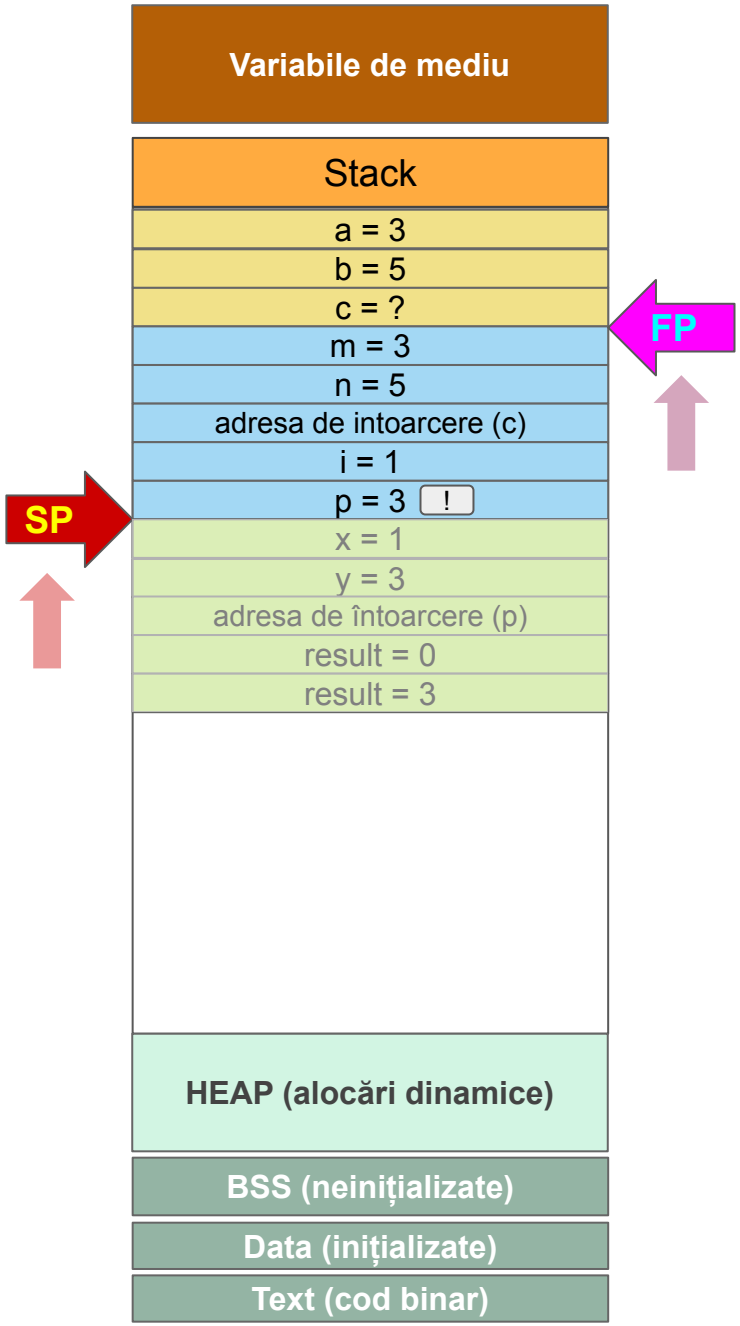


Viața unui program

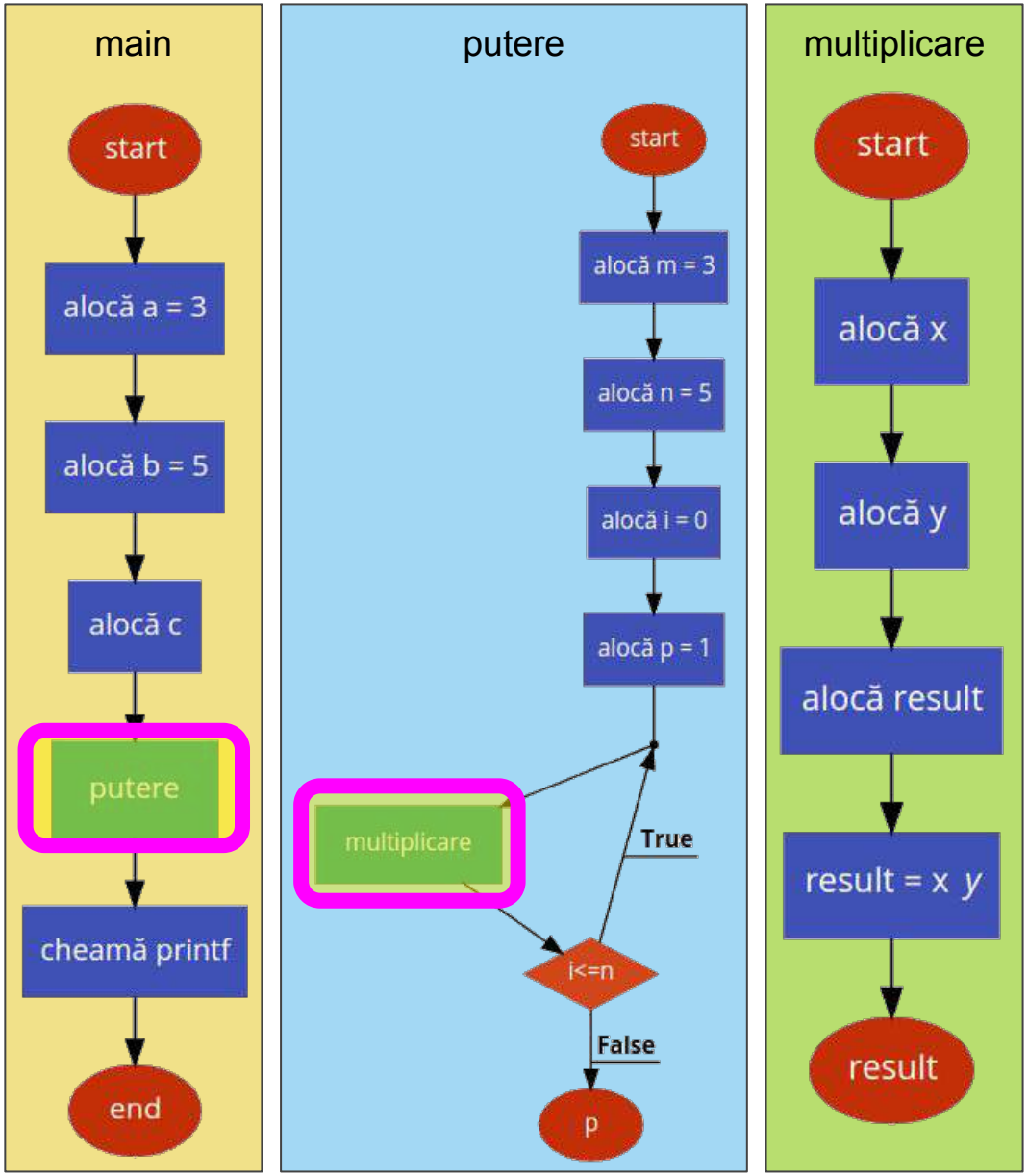


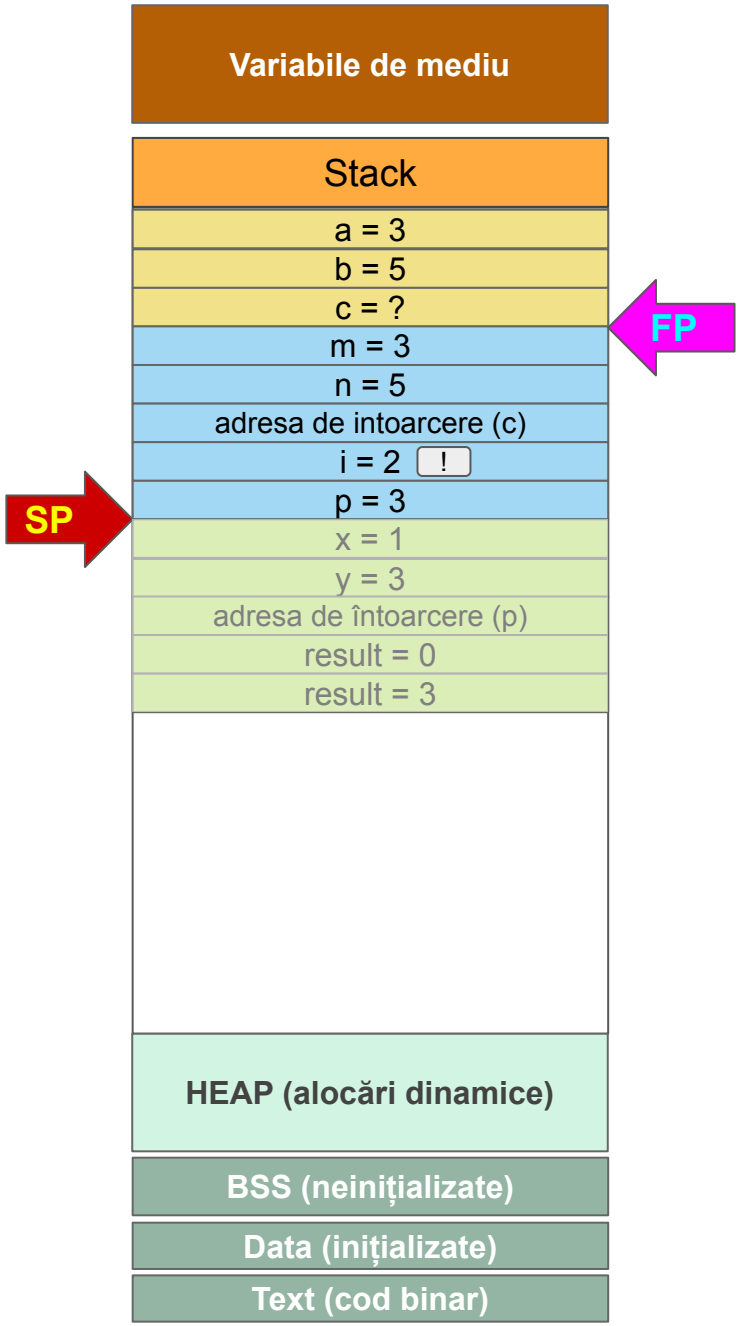




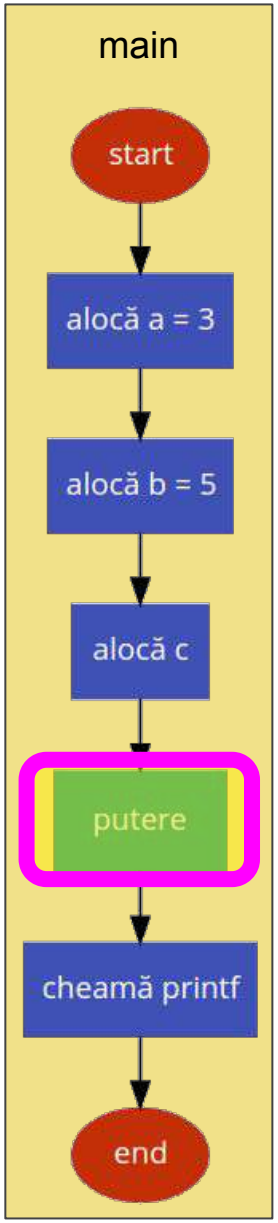


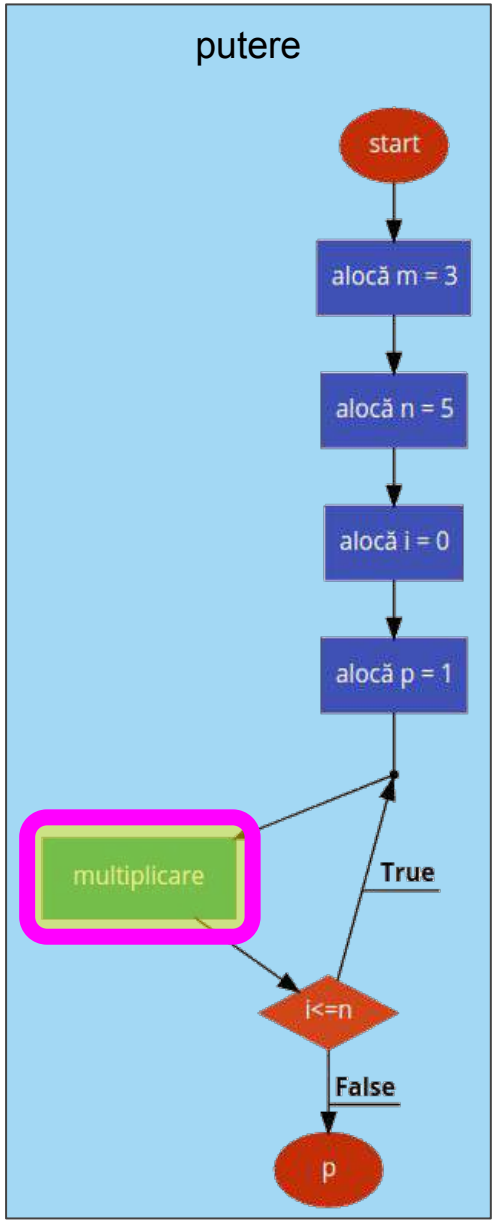
Viața unui program

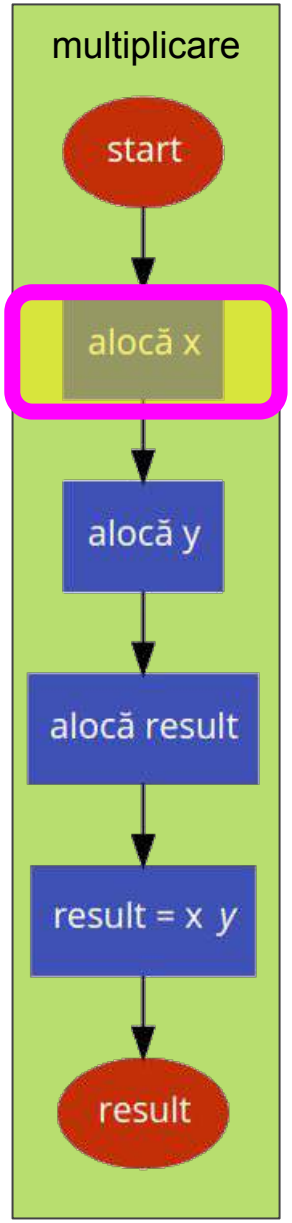


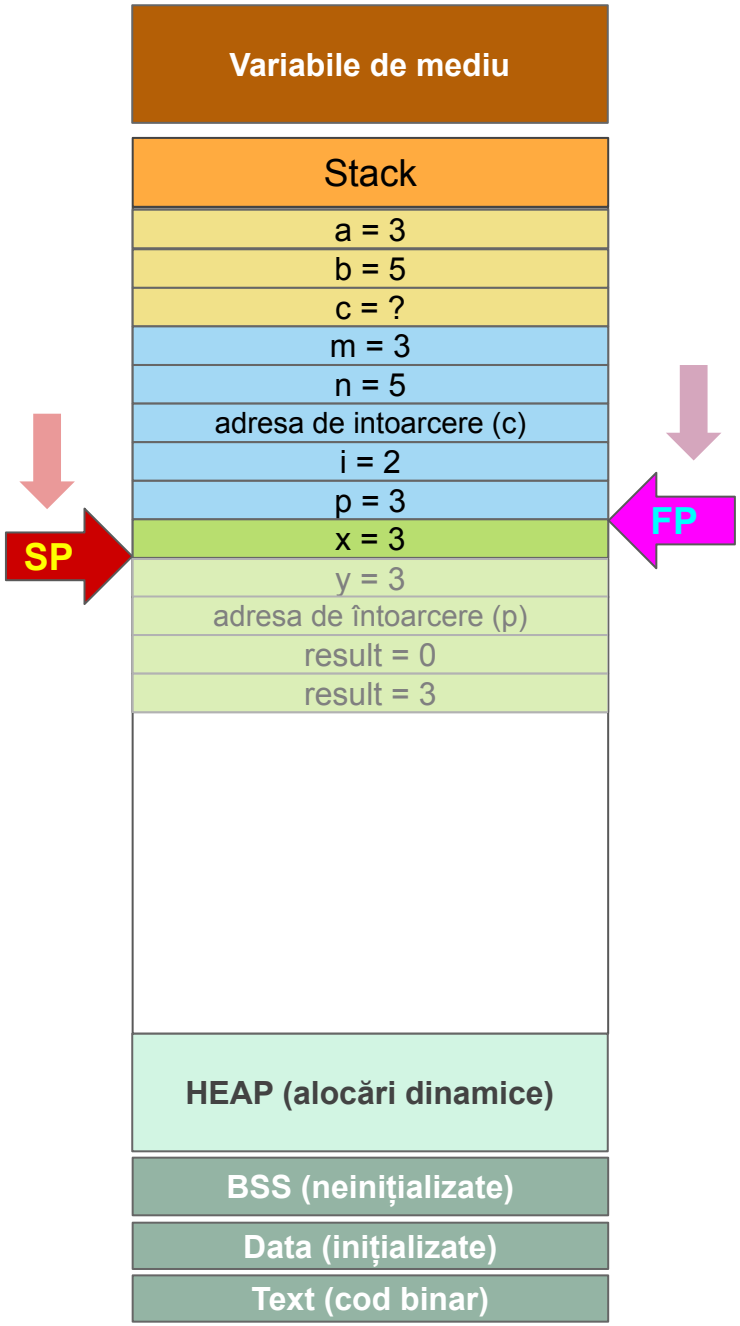


Viața unui program

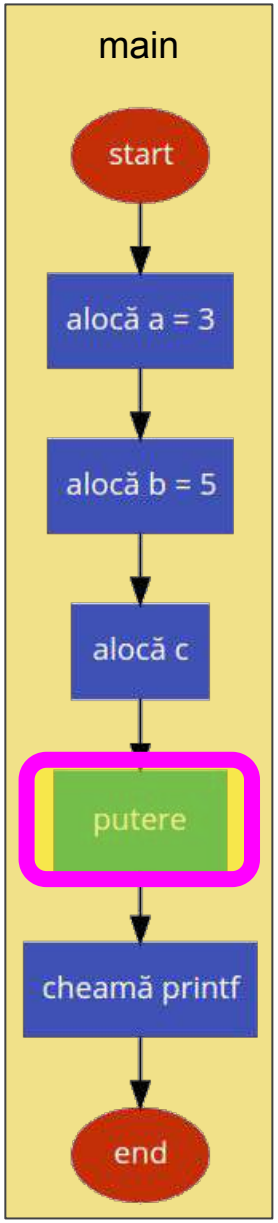


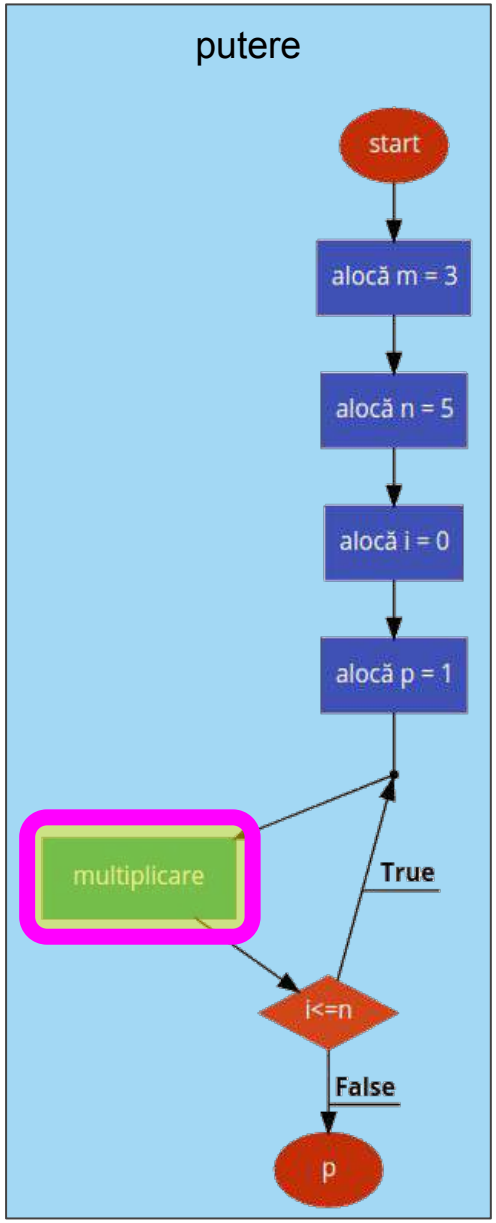


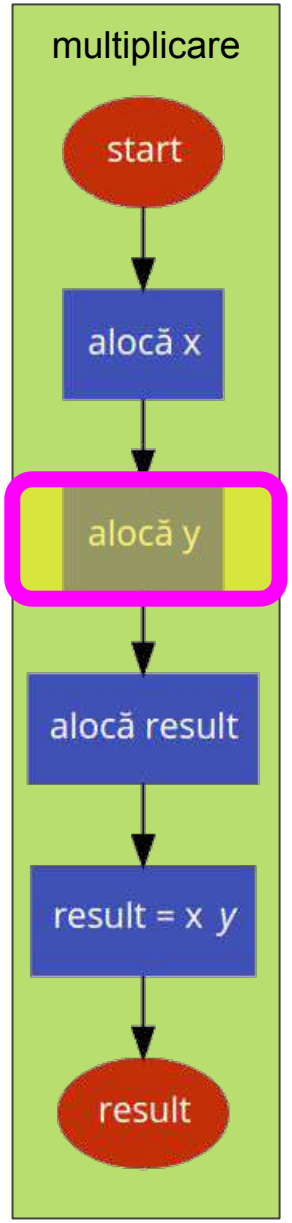


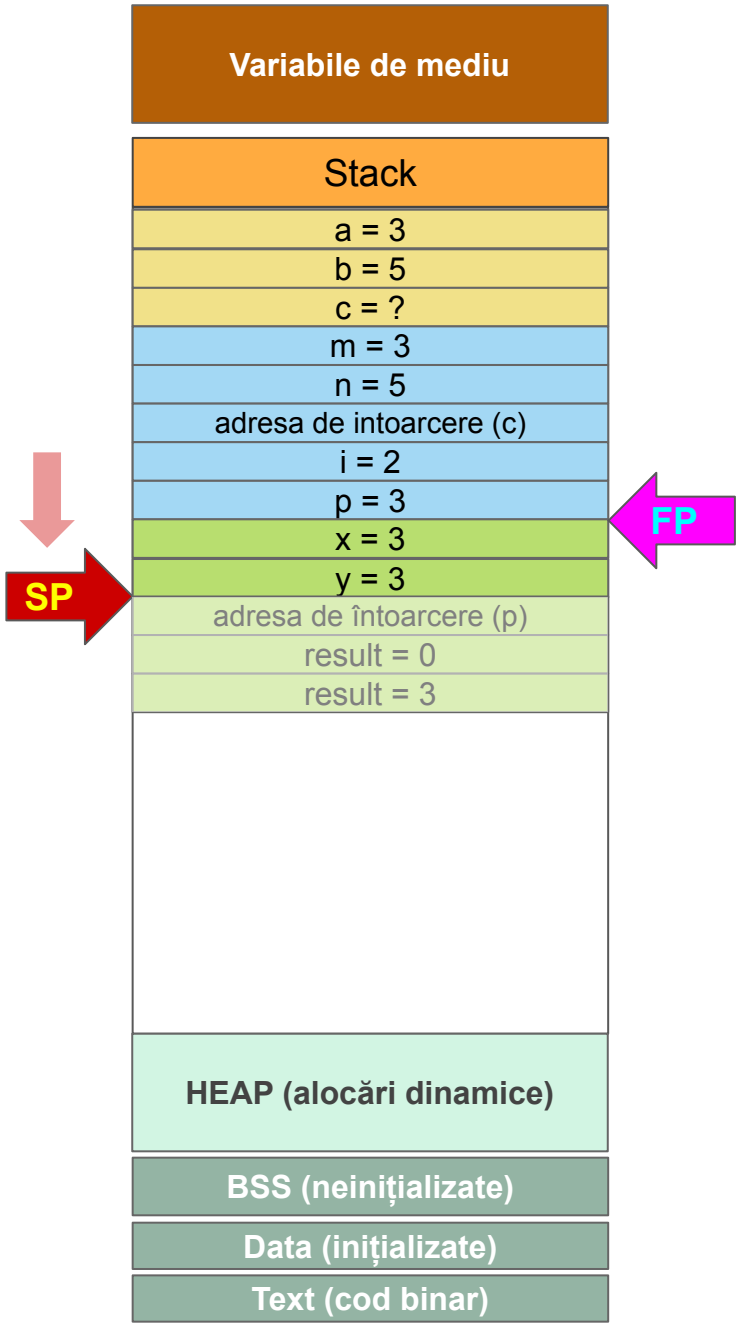


Viața unui program

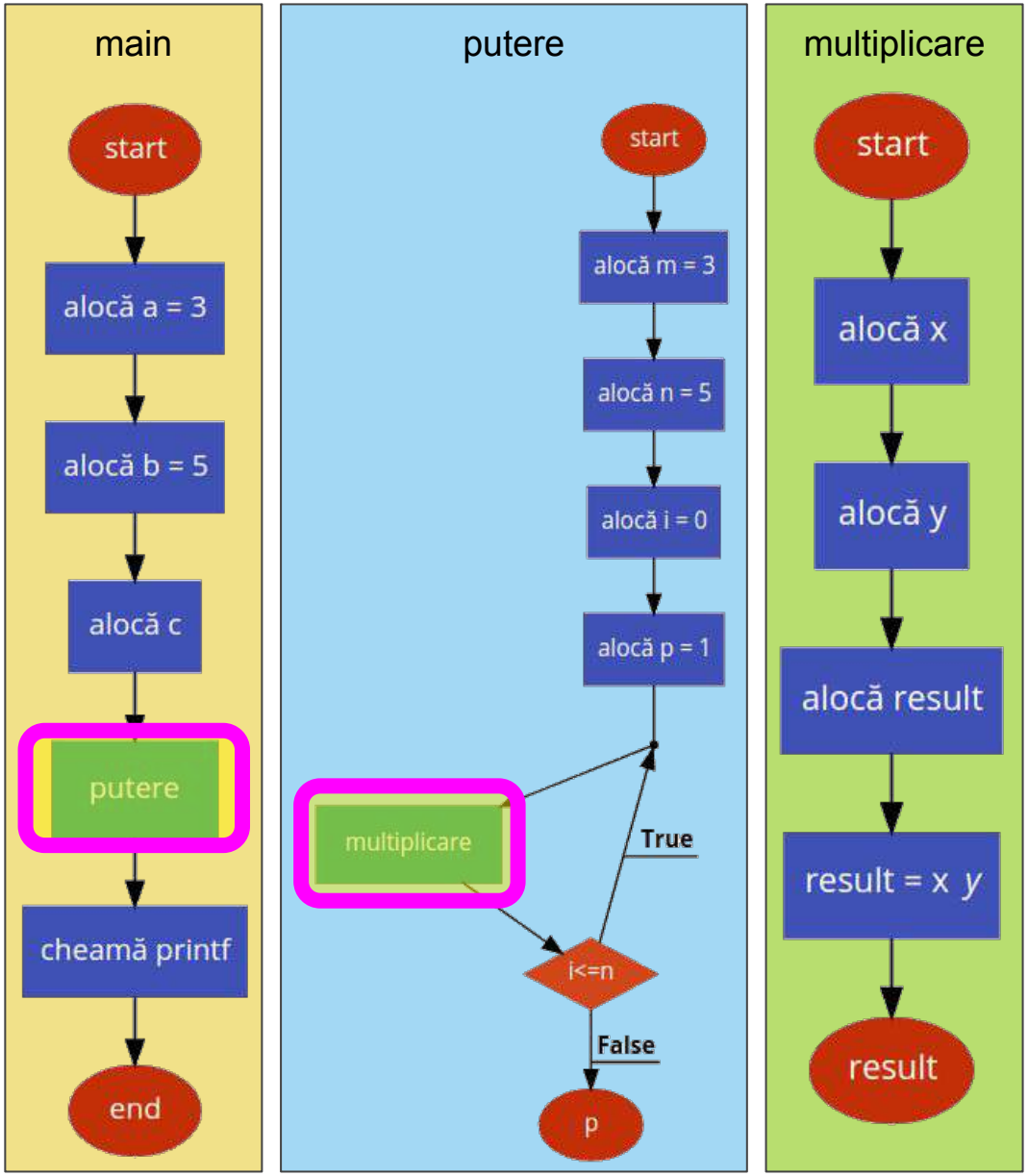


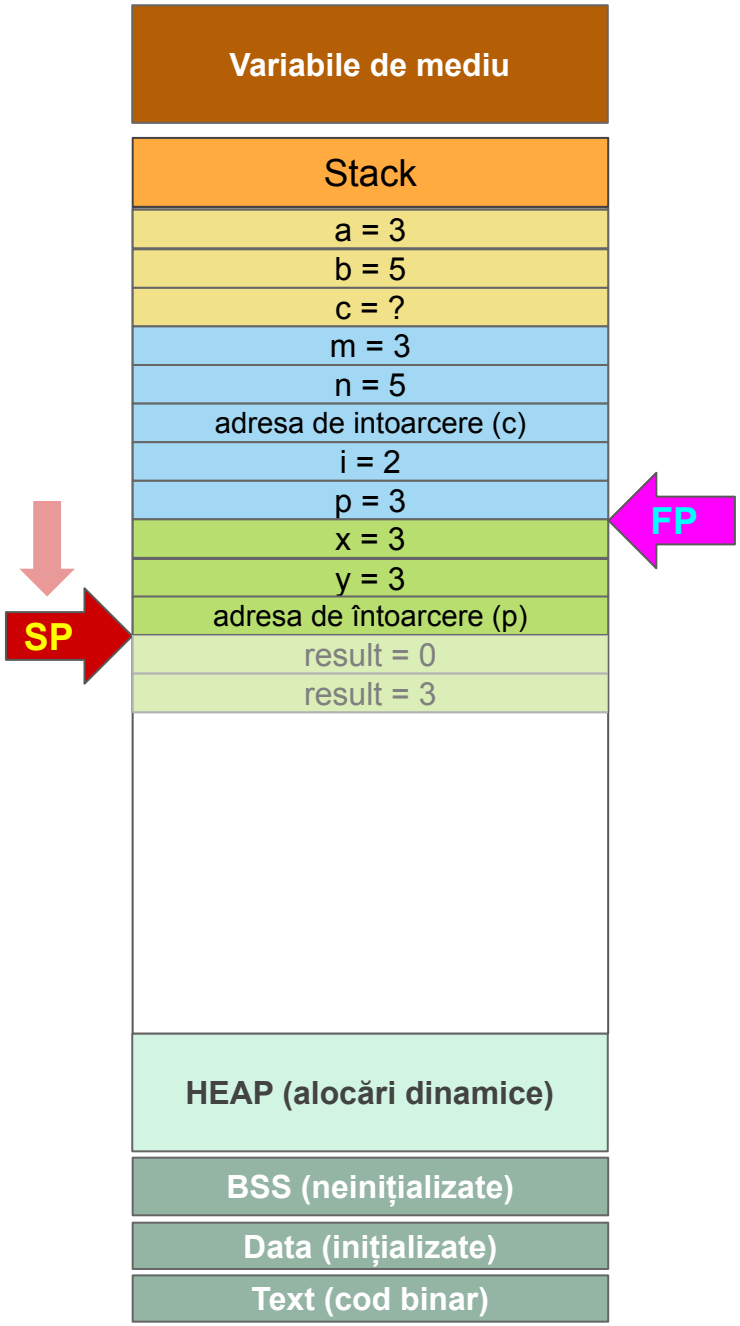




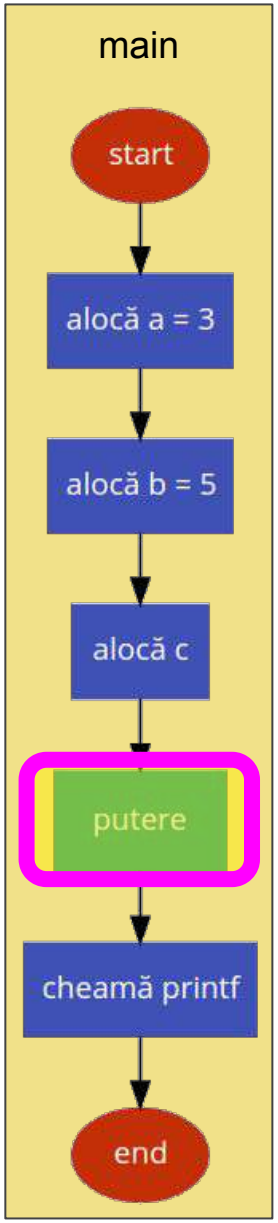


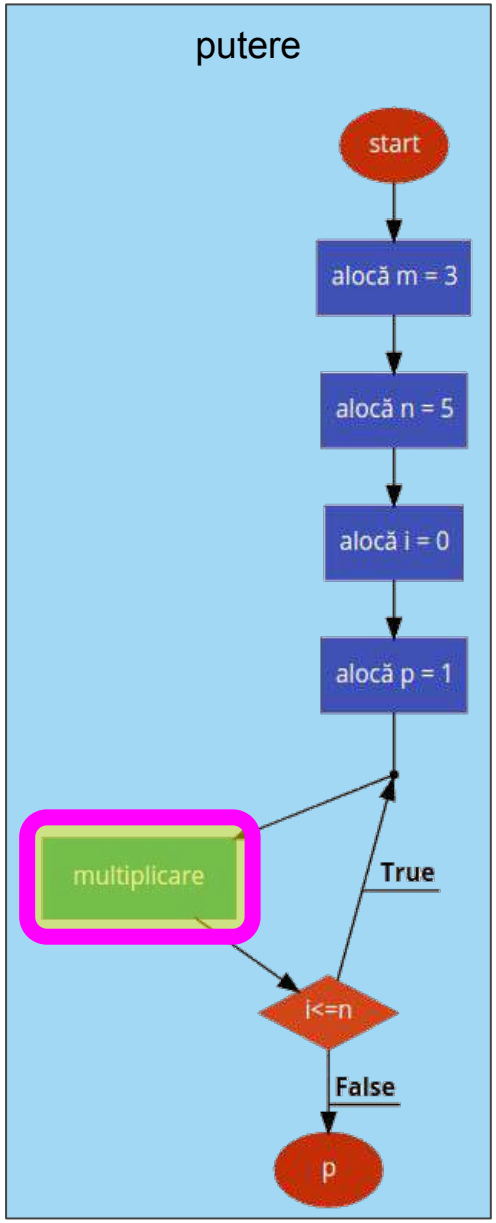
Viața unui program

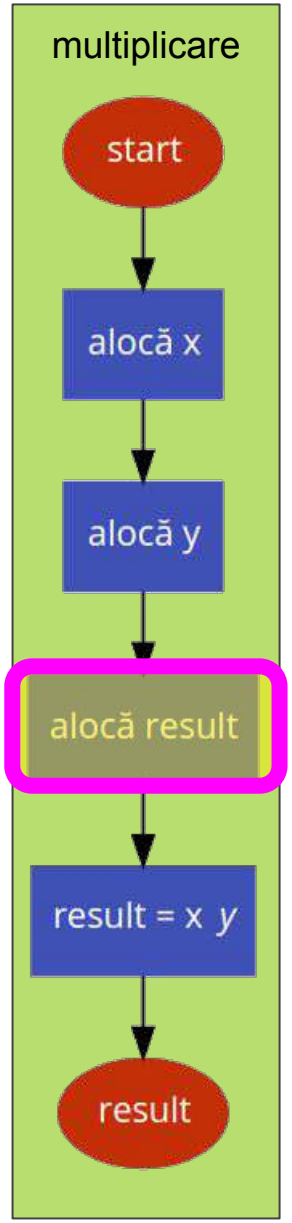


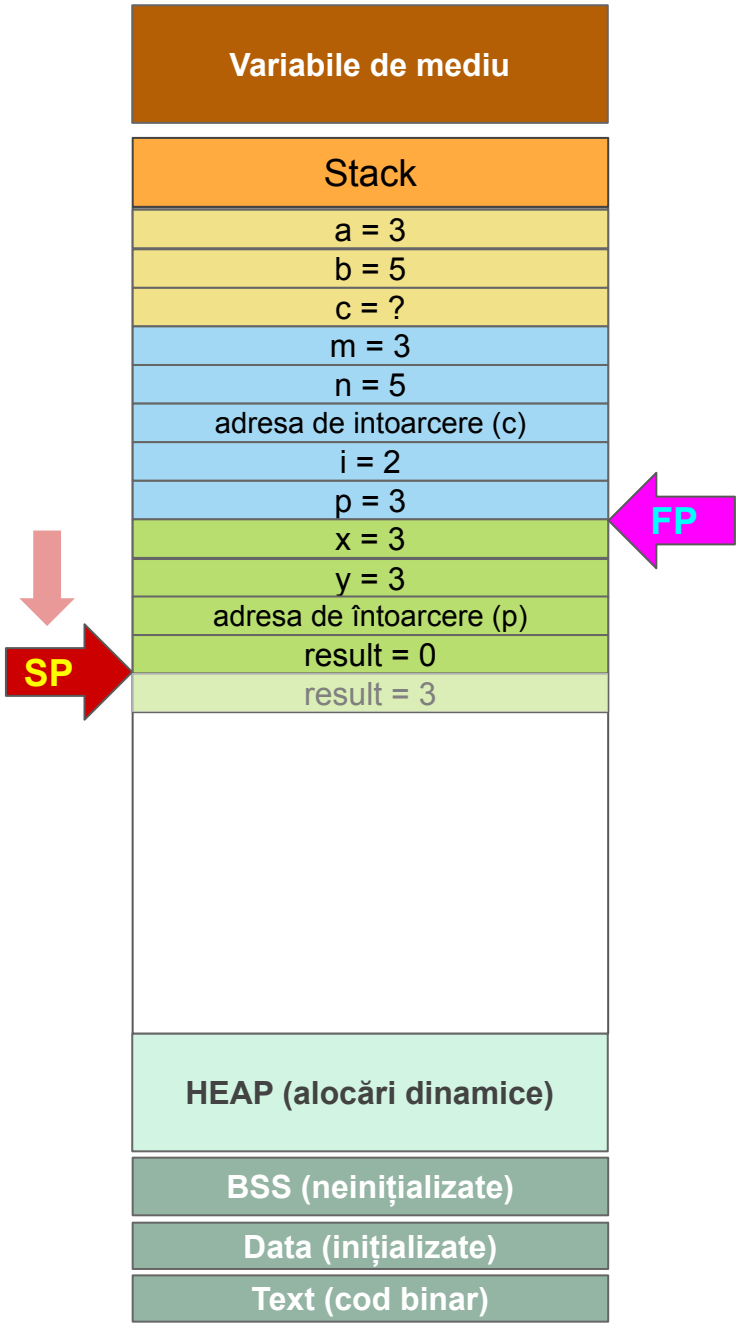


Viața unui program

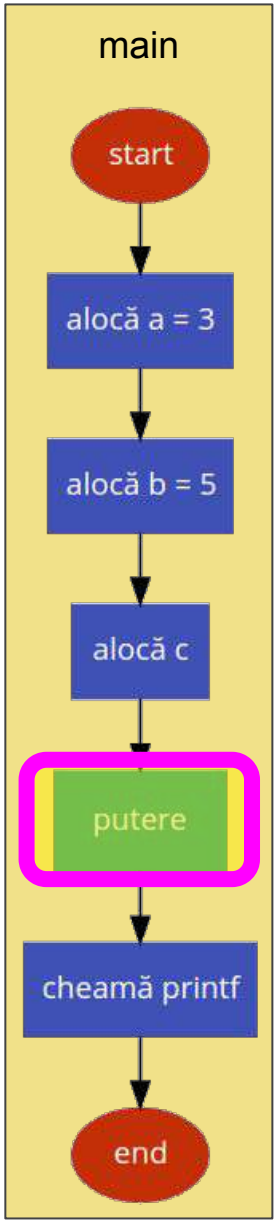


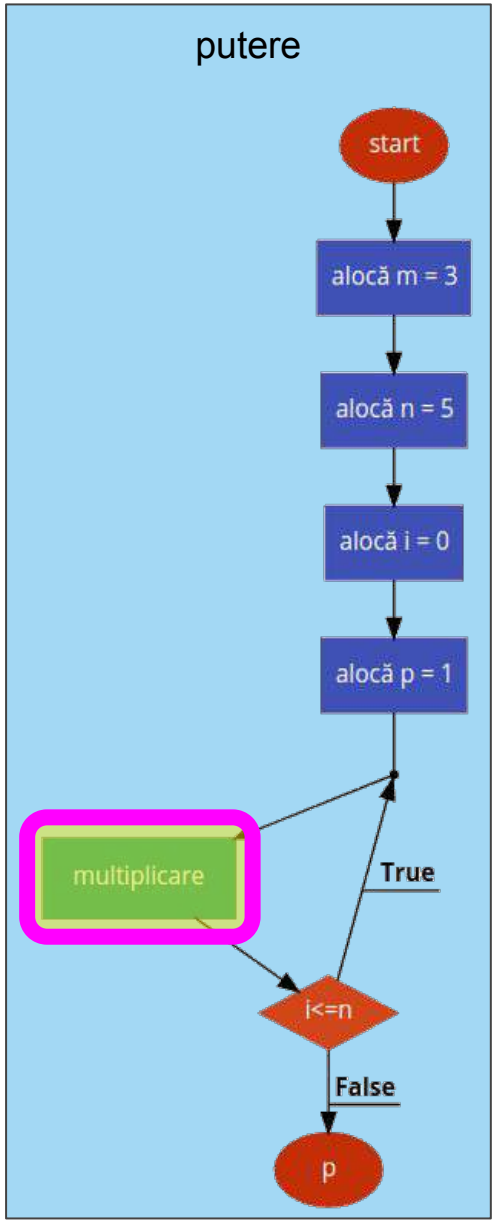


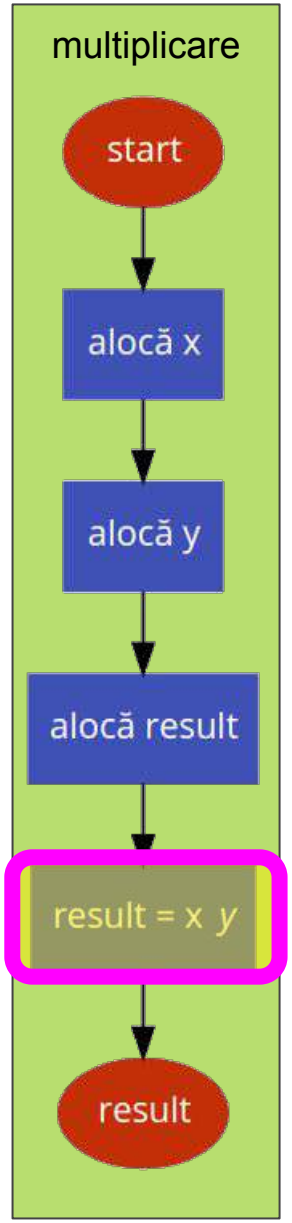


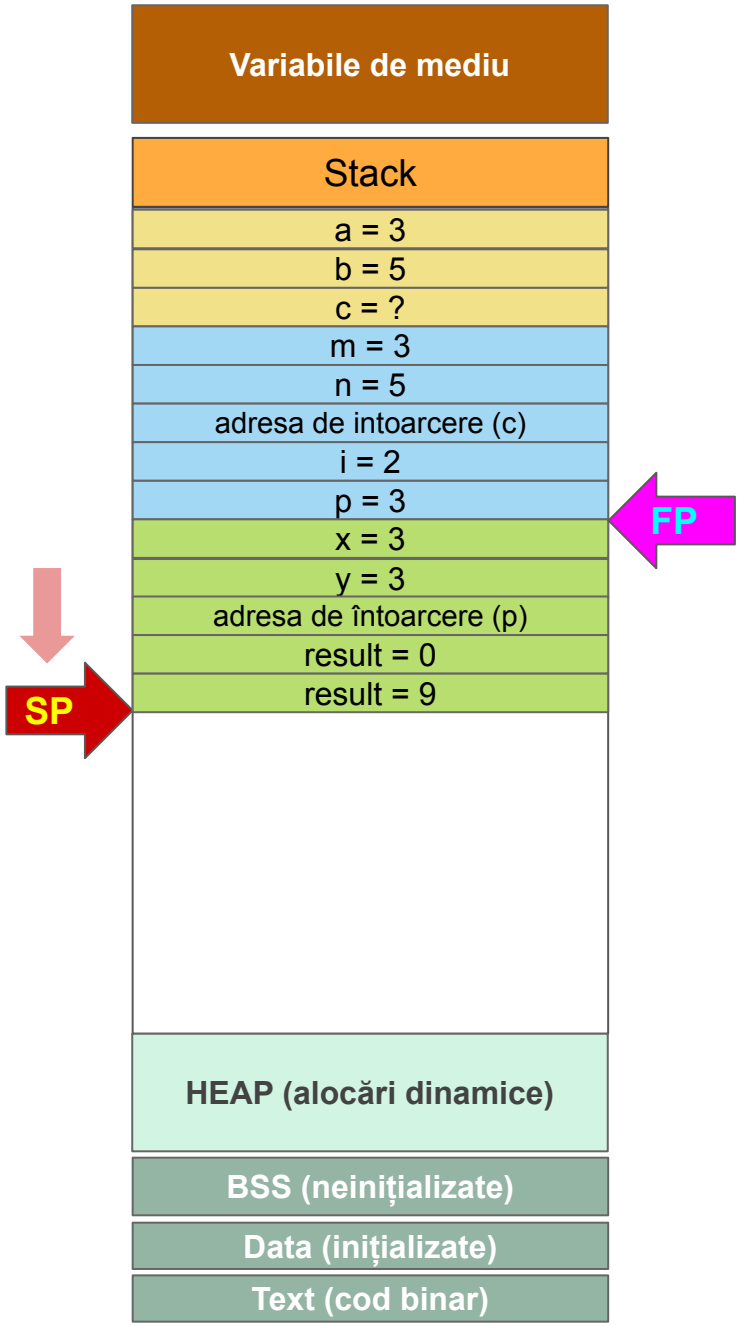


Viața unui program

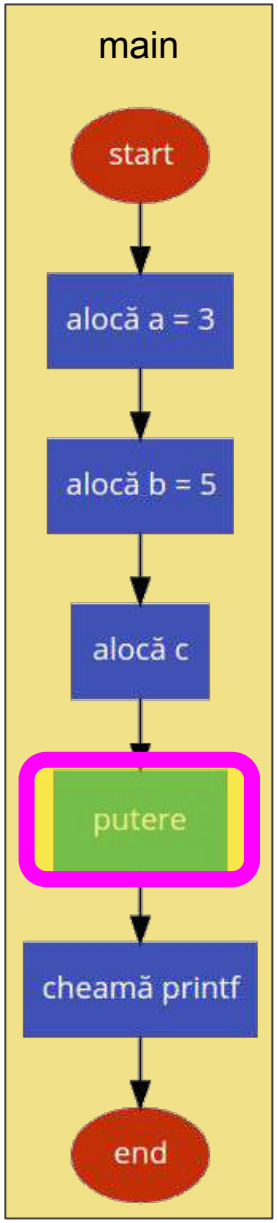


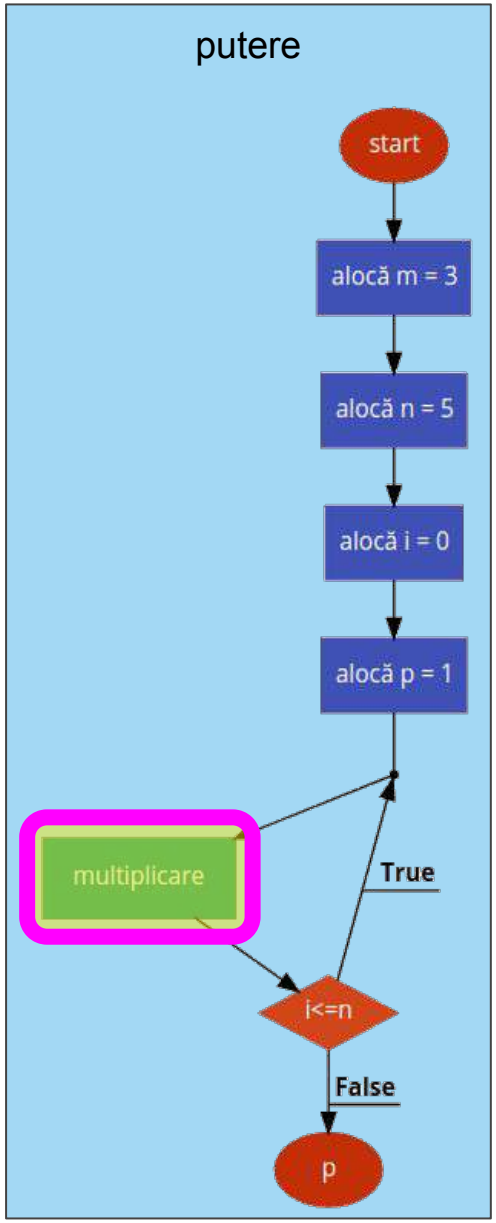


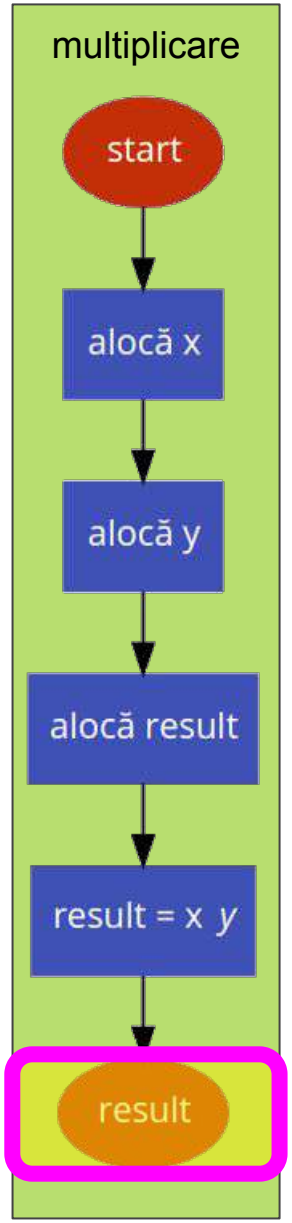


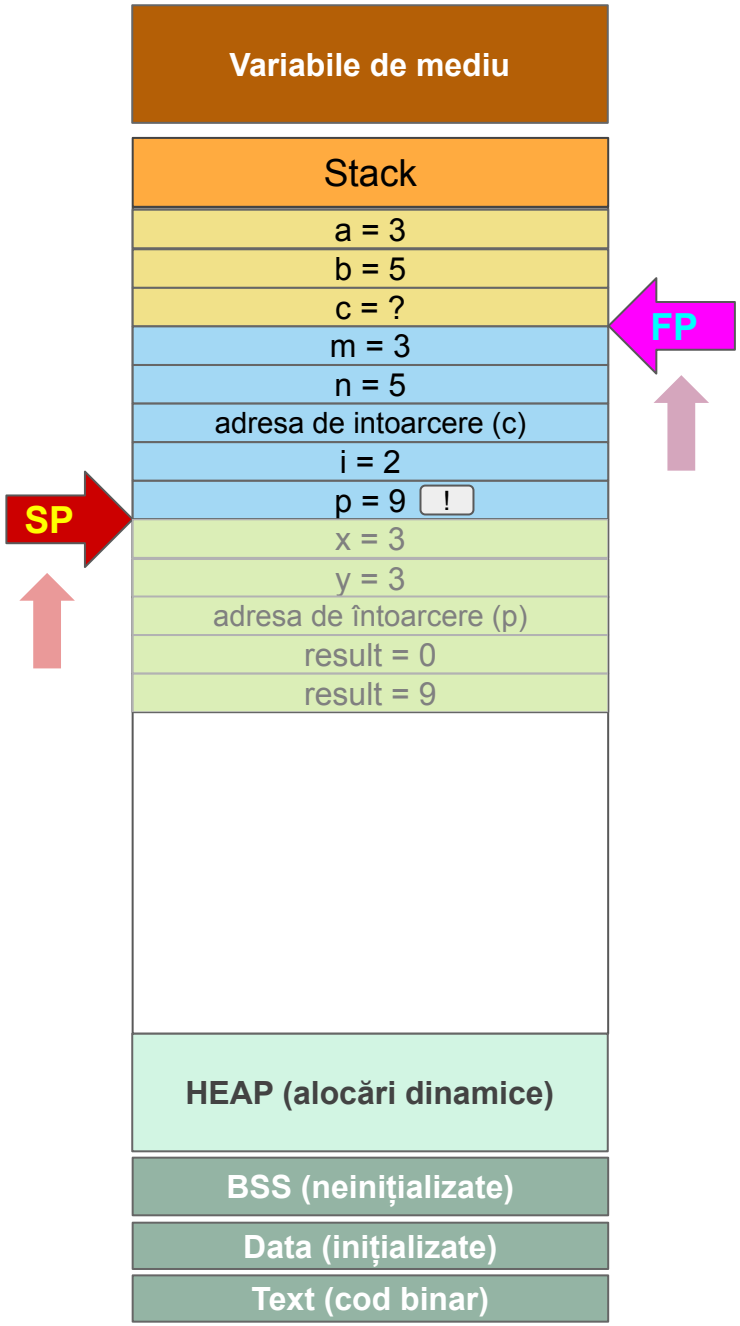


Viața unui program

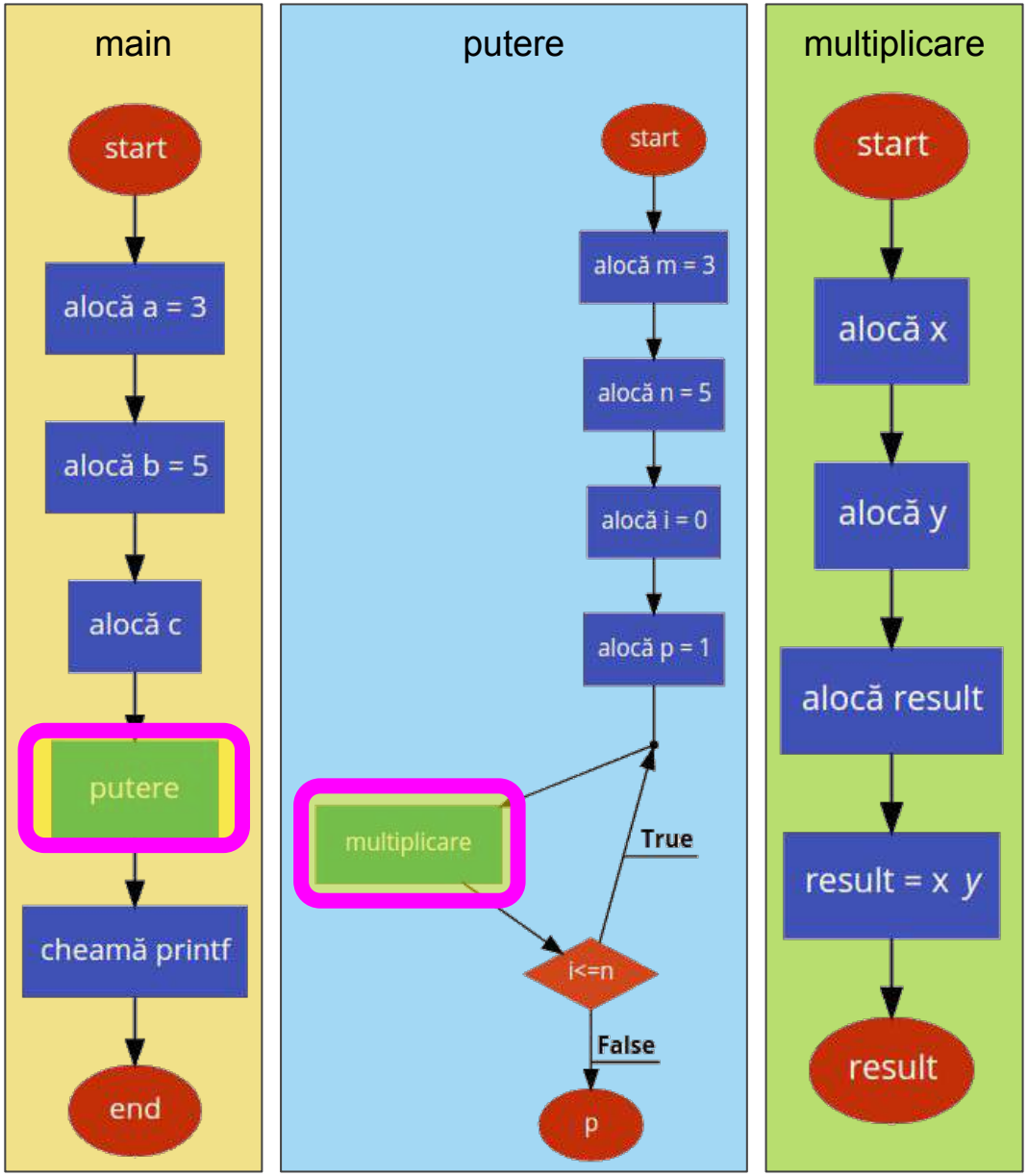


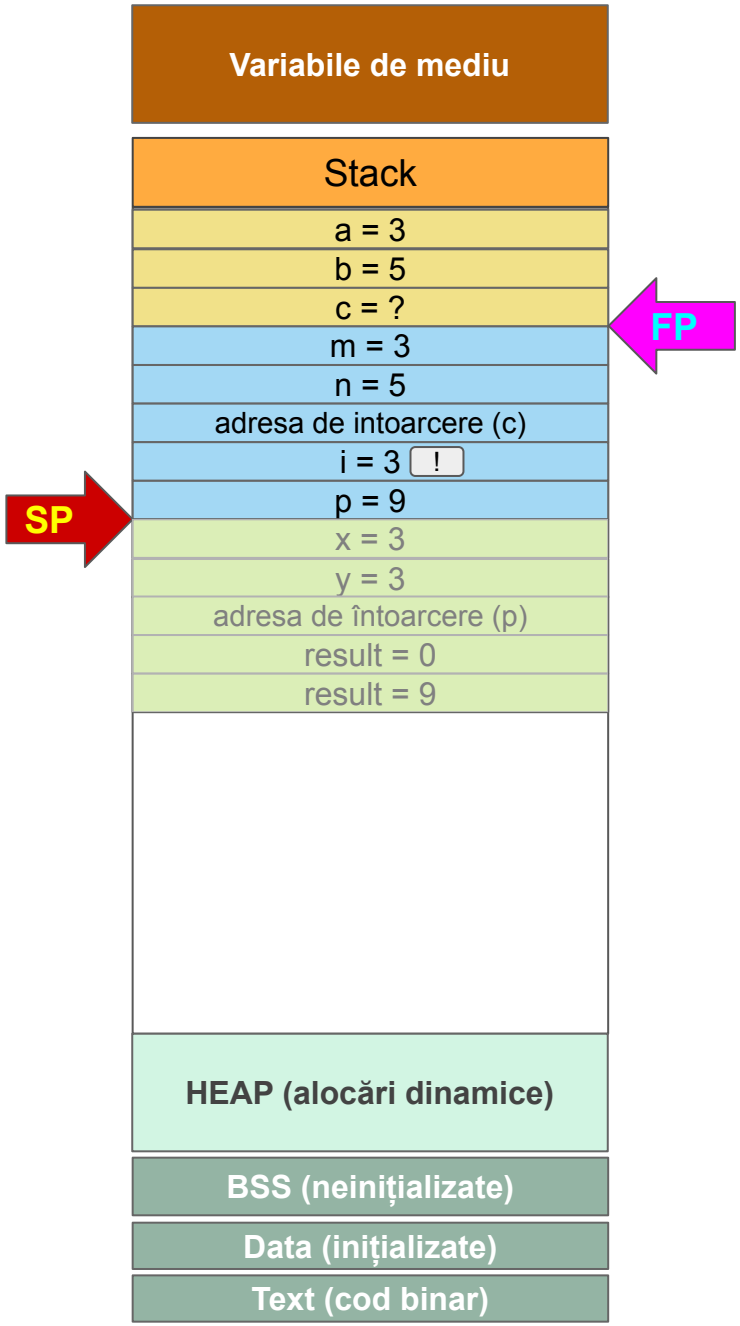






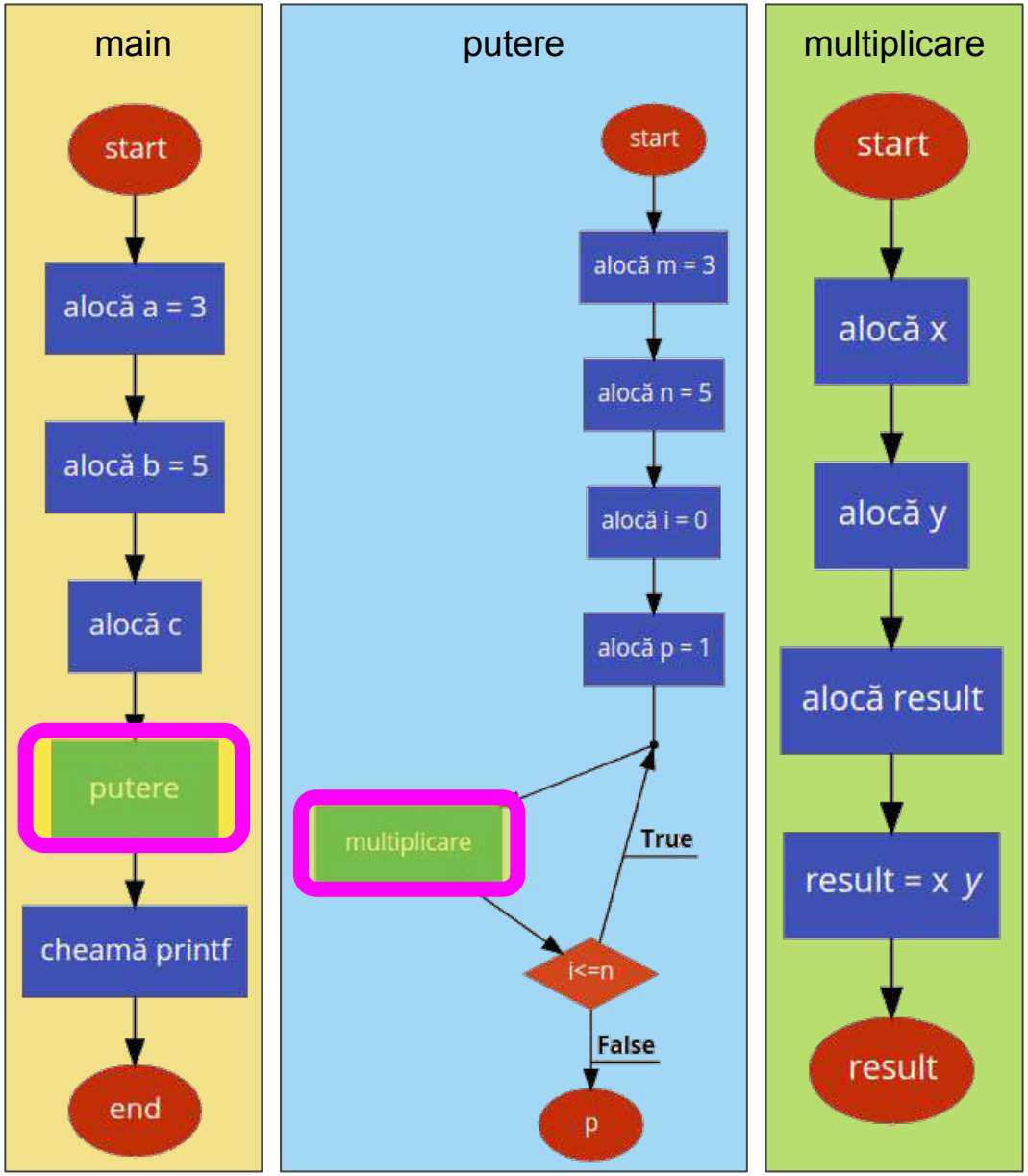
Viața unui program

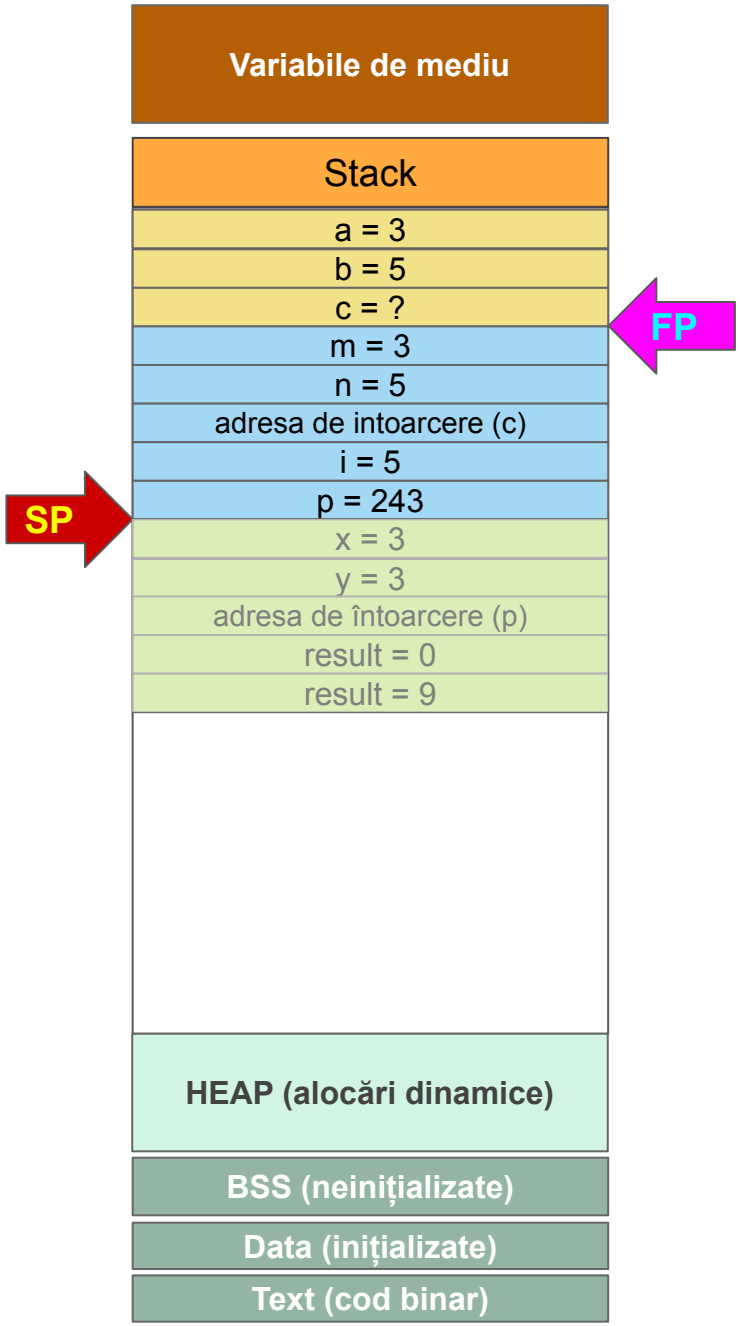




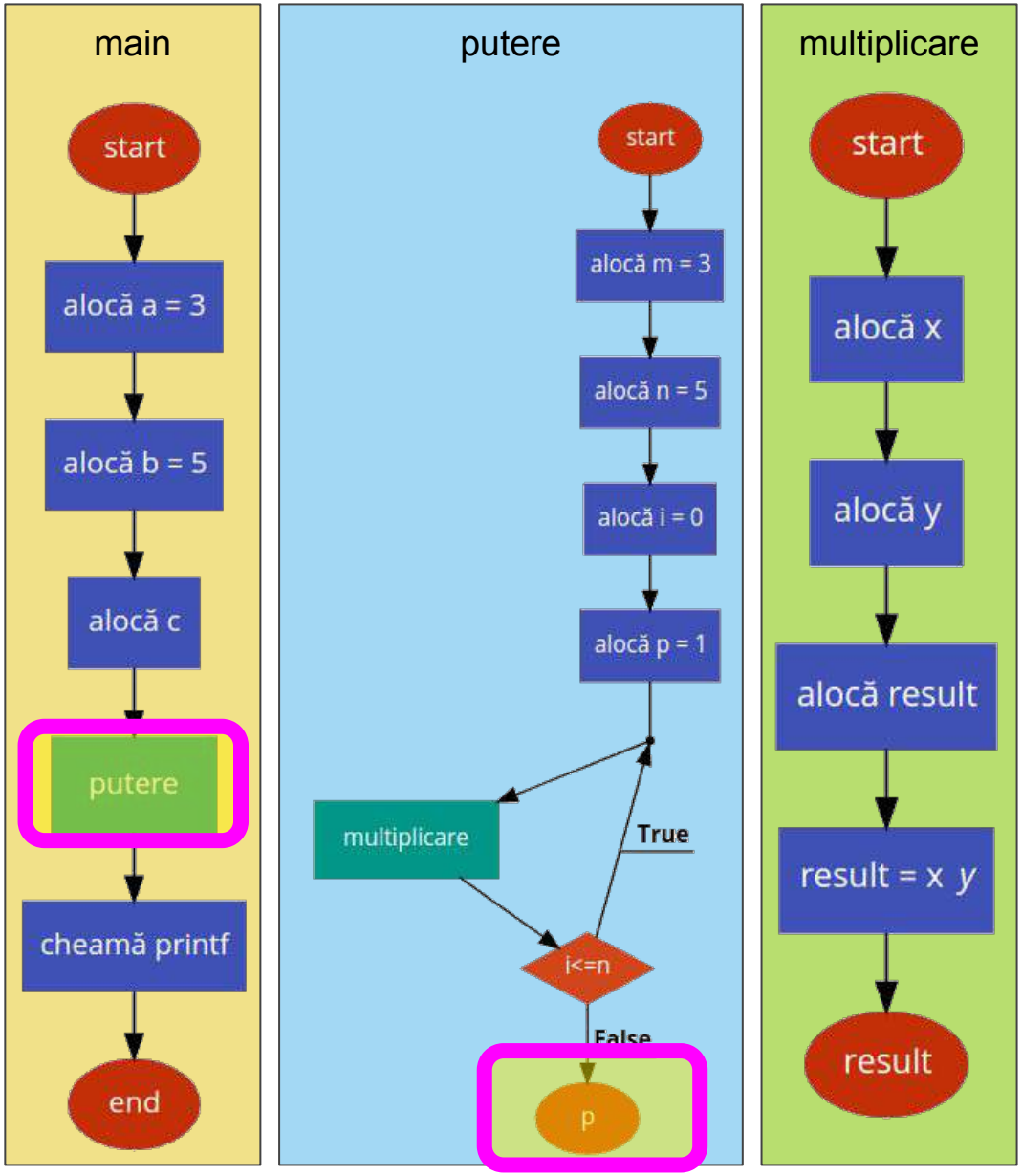
Programul continuă să ruleze până
când i devine egal cu n

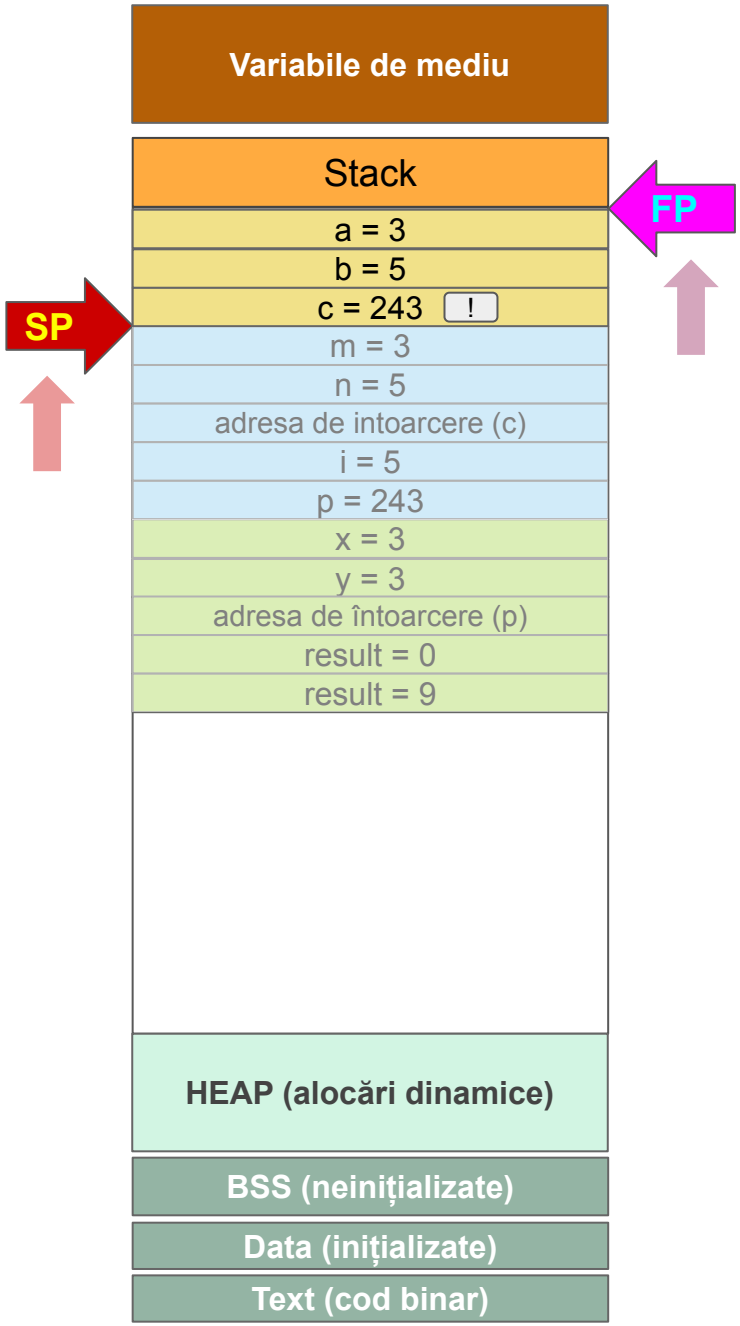
Viața unui program



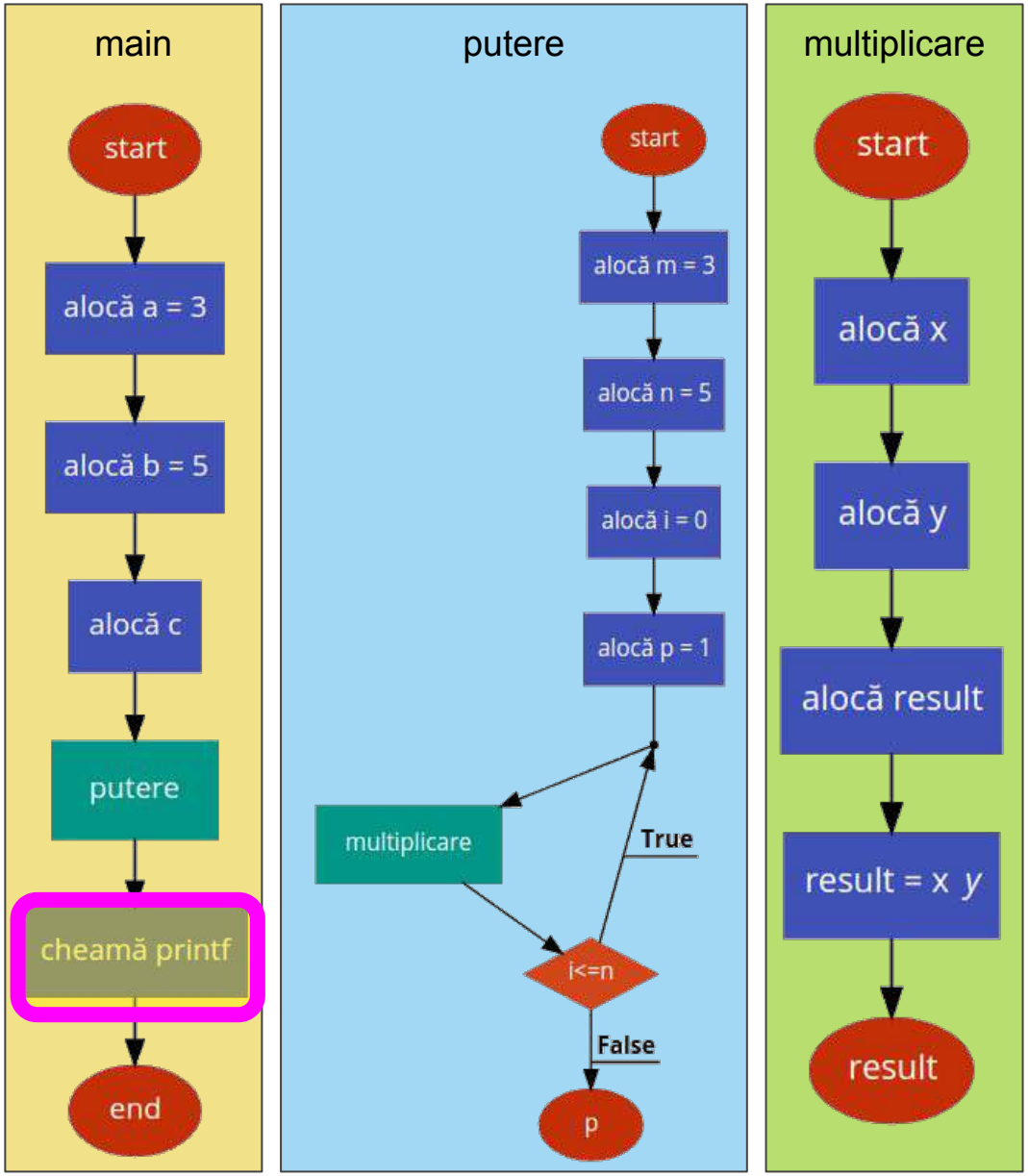


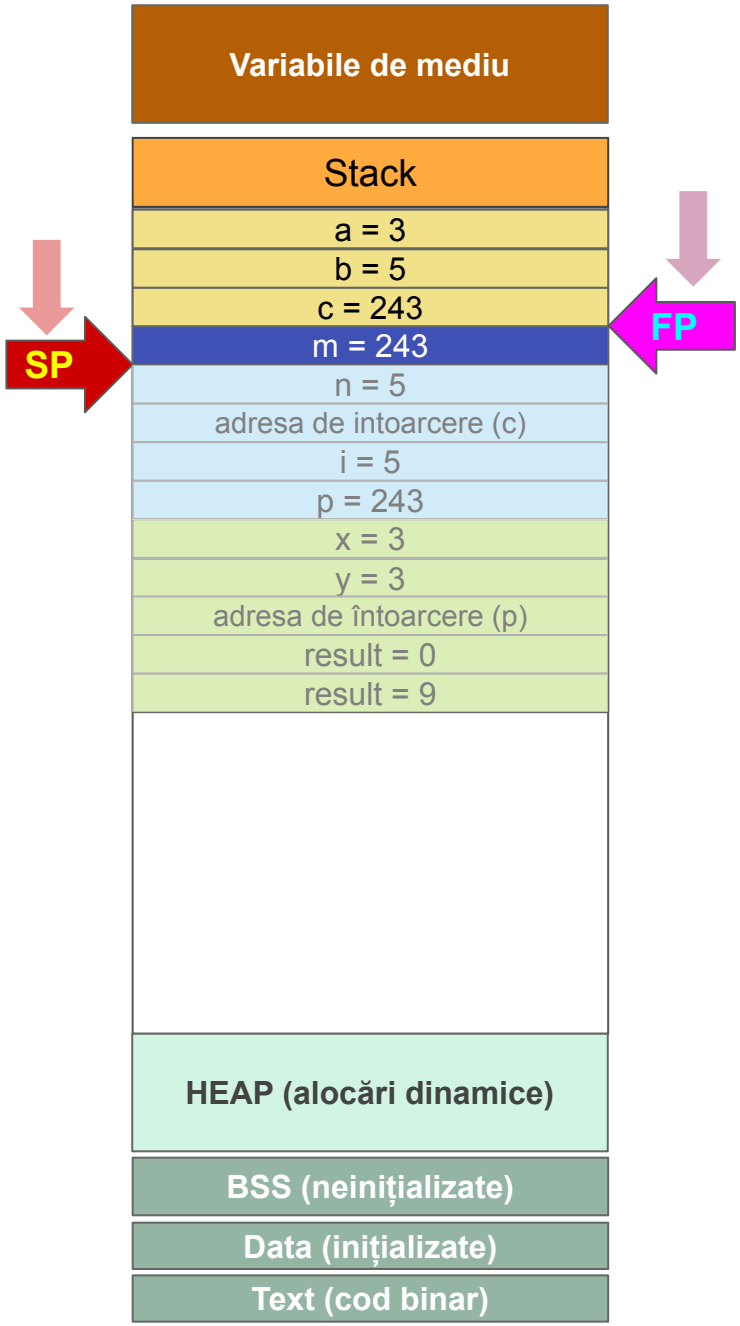
Viața unui program



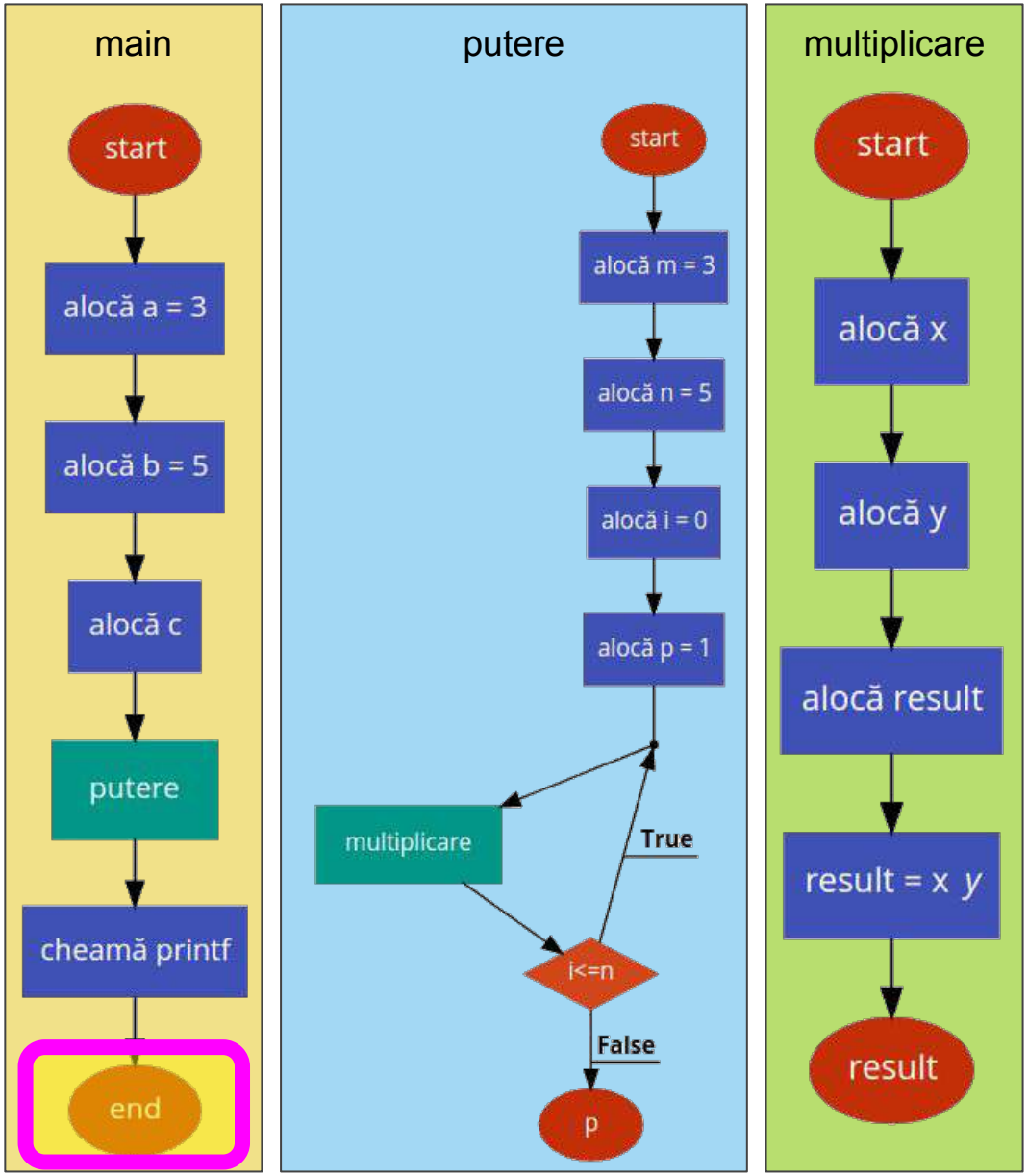


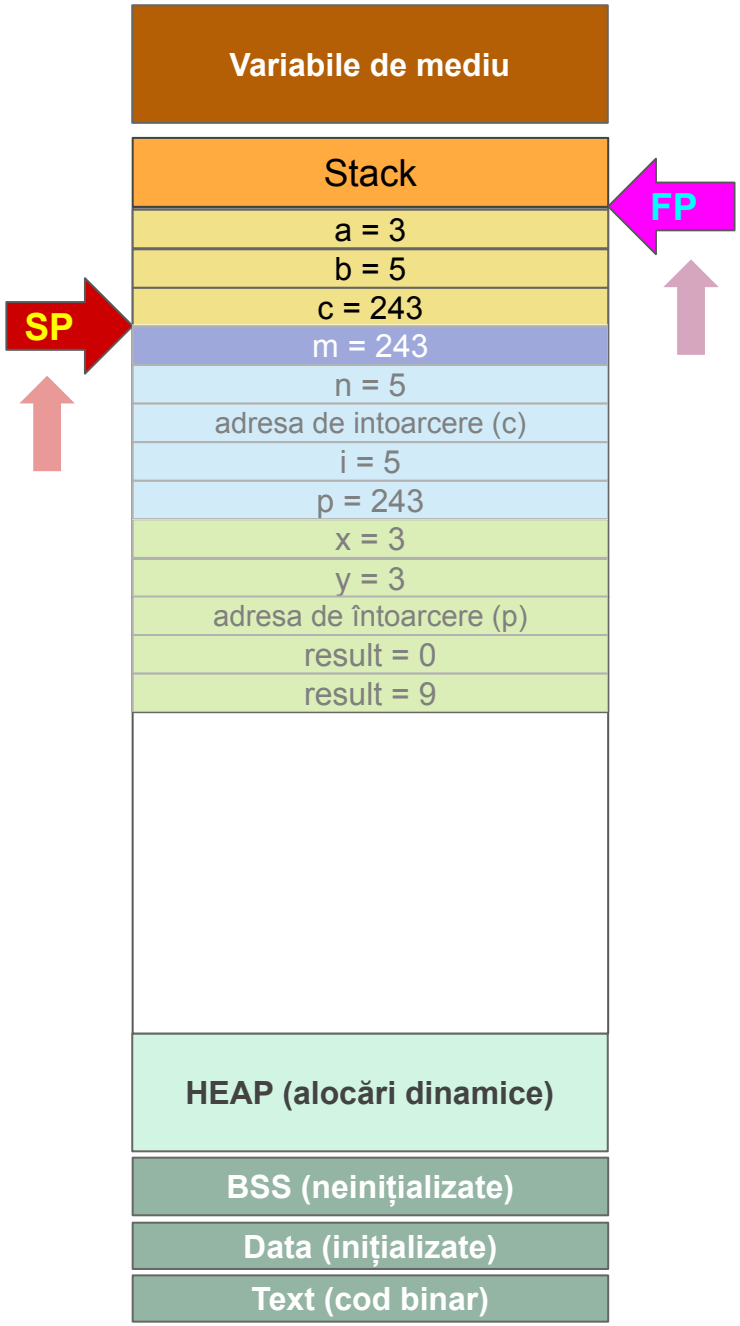
Viața unui program



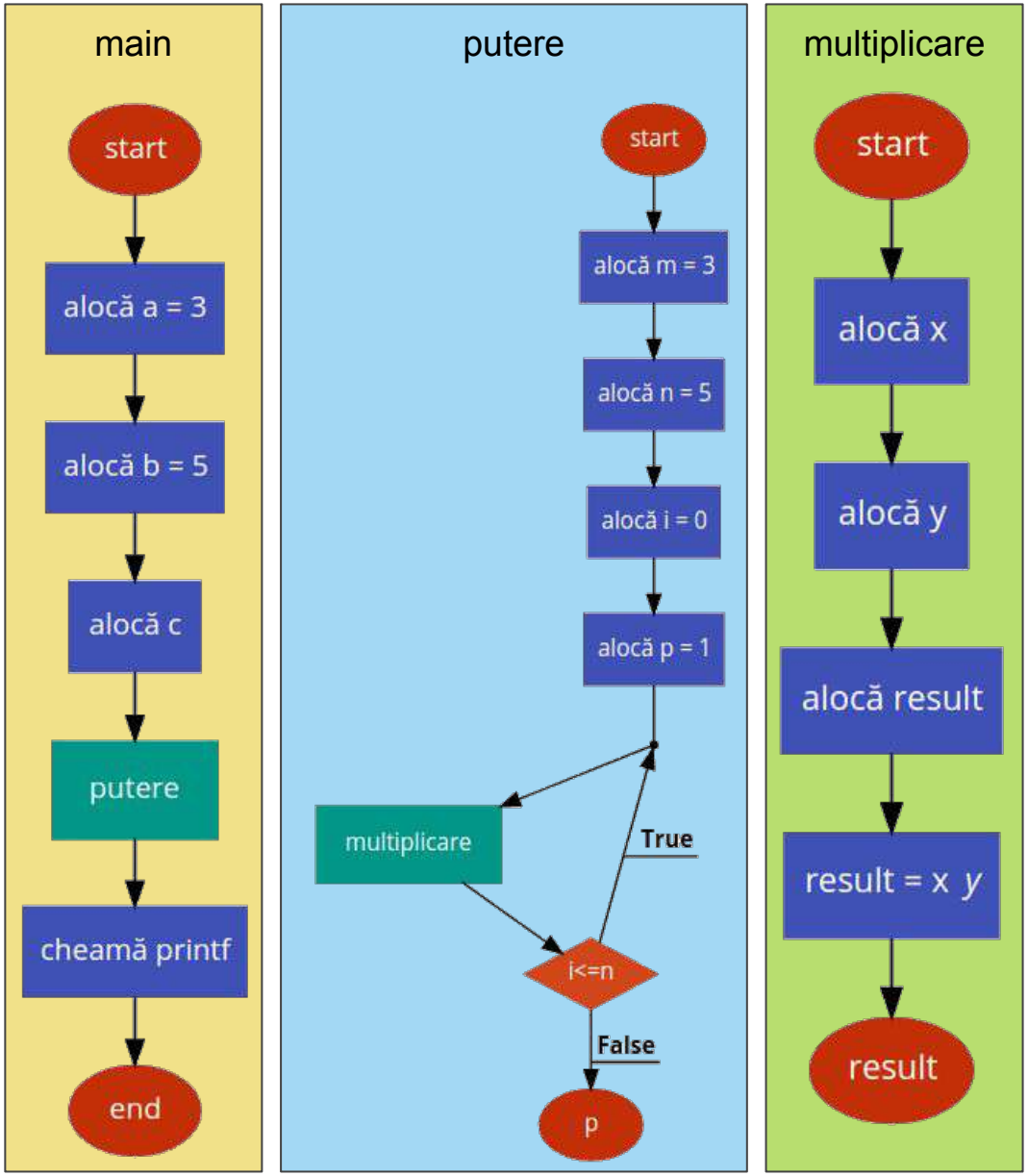


Viața unui program





Viața unui program



Ca de fiecare dată, compilatoarele moderne sunt mult mai complicate. Anumite subrutine pot să nu aibă un frame, variabilele vor fi păstrate în registrele procesorului - decizia este a compilatorului în funcție de numărul variabilelor neceare.

Frame pointerul poate fi de asemenea omis de către compilator (compilatoarele au opțiunea -fomit-frame-pointer, care face un cod cu foarte multe apeluri de subrutine puțin mai rapid. Citirea variabilelor se face în funcție de stack pointer.

De reținut că fomit-frame-pointer poate face debuggingul imposibil pe anumite platforme - debuggerele se bazează pe acesta pentru identificarea corectă a variabilelor in memorie.

Programarea structurată are suport hardware.

Programarea structurată are suport hardware. Altfel spus, setul de instrucțiuni x86 (si multe altele) constituie un procesor structural.

Unitatea de procesare este o instructiune (parte dintr-o secventă algoritmică), modul segmentat de adresare al memoriei este potrivit pentru acest gen de structură.

Procesoare “Orientate Obiect”

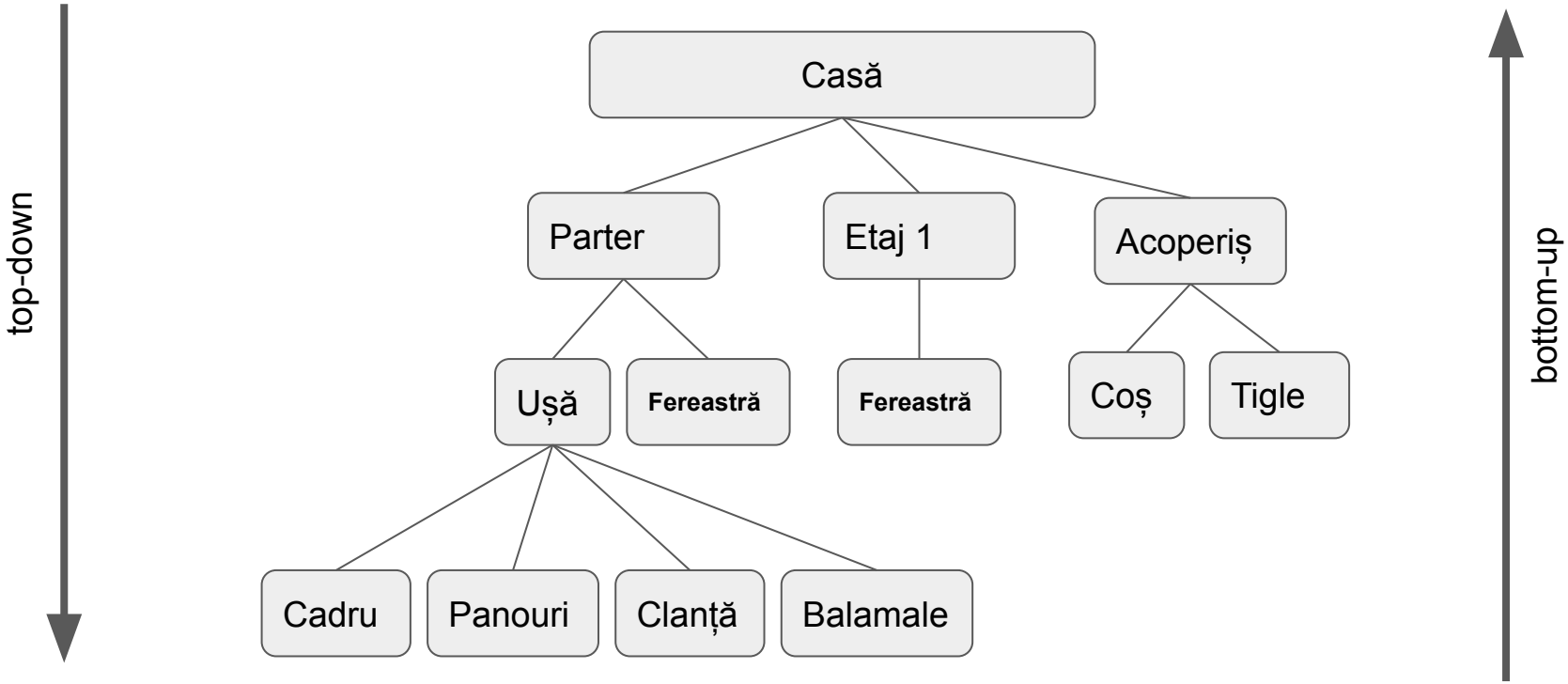
- Sunt compuse din mai multe unități (mai multe microprocesoare, fiecare cu scopul său)
- Permit adresarea memoriei sub forma unor pachete, Unitatea de management al memoriei este integrată în construcția procesorului.
- Au instrucțiuni specifice pentru accesarea segmentelor variabile de memorie
- Majoritatea implementărilor au fost eșecuri comerciale.

https://en.wikipedia.org/wiki/Intel_iAPX_432
https://en.wikipedia.org/wiki/Rekursiv
http://merlintec.com/lsi/tachyon/
https://www.researchgate.net/publication/224163056_An_object-oriented_processor_with_Just-In-Time_compilation_support/link/546d3e0f0cf26e95bc3cafd6/download

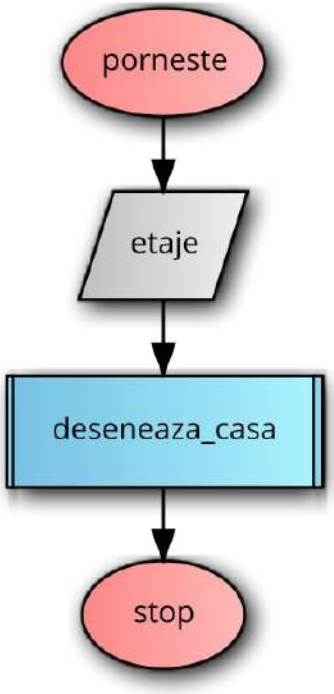
Proiectare: Top-down vs. bottom-up

Proiectarea unui program înseamnă **înțelegerea completă a unei probleme**. Pentru problemele complicate acest lucru nu se poate întâmpla dintr-un singur pas. De obicei, există două categorii de abordări: inductive și deductive.

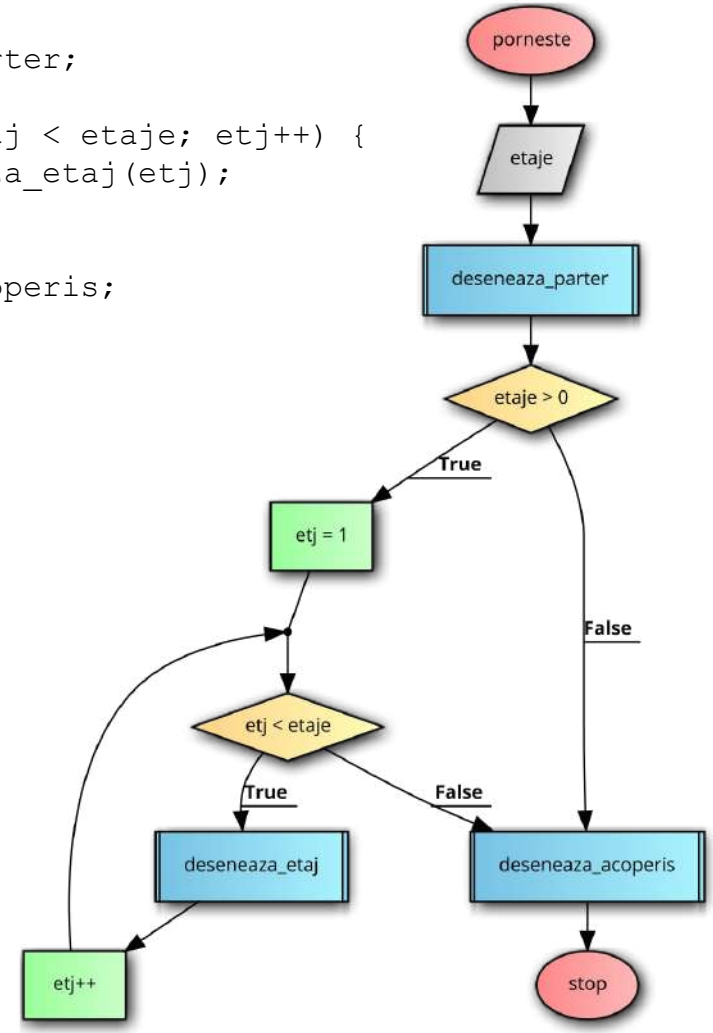
În cazul proiectării programelor, abordările se numesc **top-down** (deductiv) sau **bottom-up** (inductiv).



Top-down design

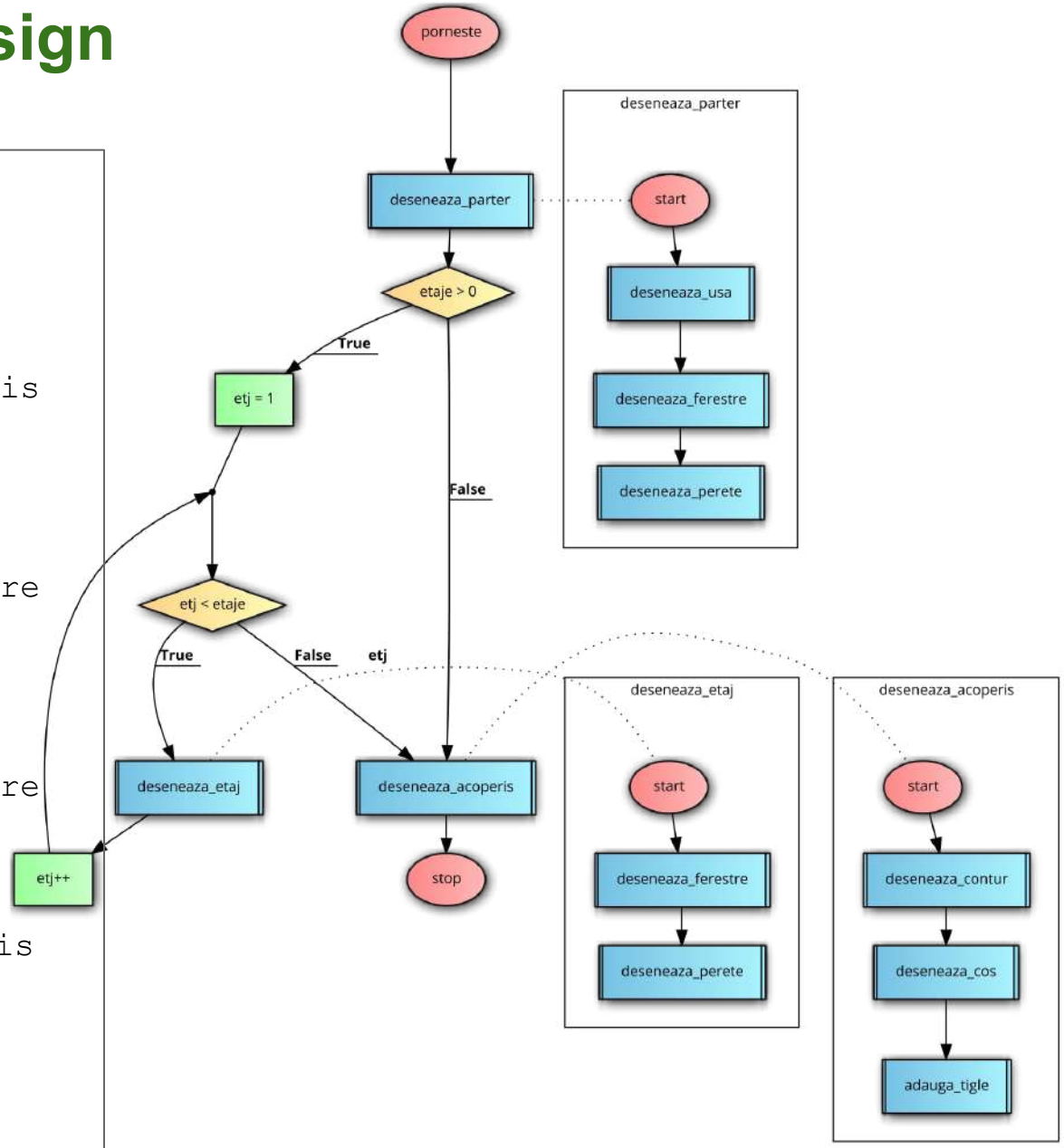



```
porneste;
/etaje/;
call deseneaza_parter;
if (etaje > 0) {
  for (etj = 1; etj < etaje; etj++) {
     call deseneaza_etaj(etj);
  }
}
call deseneaza_acoperis;
stop;
```



Top-down design

```
read etaje
call deseneaza_parter
do
    call deseneaza_etaj
while count <= etaj
    call deseneaza acoperis
end

function deseneaza_parter
    call deseneaza_usa
    call deseneaza ferestre
    call deseneaza_perete
end function

function deseneaza_etaj
    call deseneaza ferestre
    call deseneaza_perete
end function

function deseneaza_acoperis
    call deseneaza_contur
    call deseneaza cos
    call adauga_tigle
end function
```



7. Limbajul de programare C

Limbajul de programare C

- Un limbaj cu scop general
- Un limbaj care folosește în principal programarea imperativă, structurată (procedurală)
- Tipurile de variabile sunt declarate și verificate înainte de compilare.

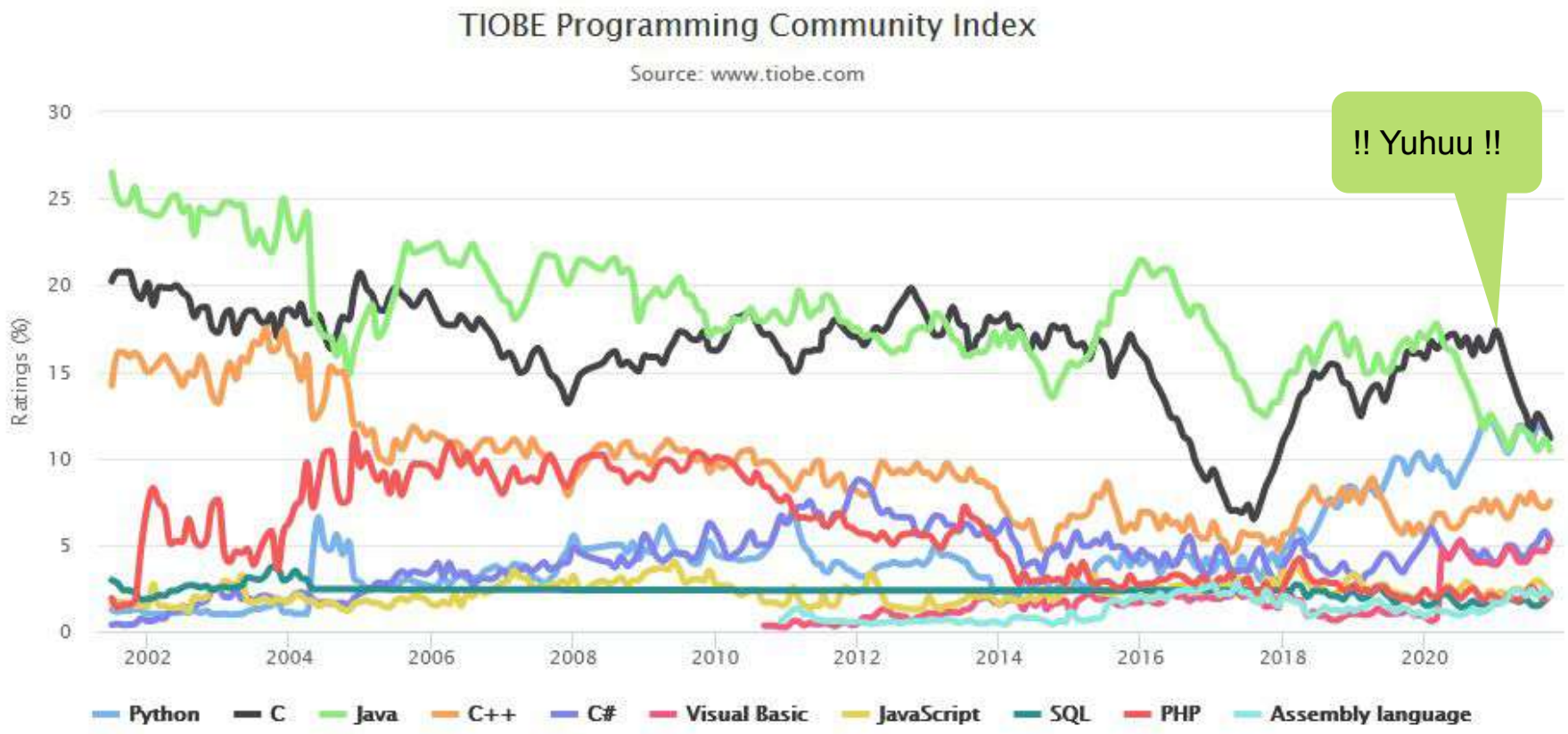
- Un limbaj relativ simplu, destul de apropiat de limbajul de asamblare
- Un limbaj robust care permite scrierea unui program oricât de complex.
- Programele scrise în C sunt de obicei rapide și eficiente (depinde de programator)
- Un limbaj extrem de portabil (codul poate fi ușor mutat de pe o arhitectură pe alta)
- Un limbaj care se poate extinde prin gruparea unor funcții în biblioteci.
- Un limbaj care permite accesul direct la memorie (**programmer driven memory management**).

În utilizarea curentă, programele scrise în C sunt cele care sunt apropiate de hardware: nuclee ale sistemelor de operare, driverele diverselor componente hardware, nucleele jocurilor care folosesc accelerații hardware, software care rulează pe sisteme automate… etc.

De ce să învățăm C?

C este unul dintre cele mai populare limbaje de programare

Tiobe măsoară popularitatea în motoarele de căutare, nu anunțurile de angajare!!



!! Yuhuu !!

https://www.tiobe.com/tiobe-index/

De ce să cunoaștem C?

- Este rezonabil de popular, există în continuare cerere serioasă pentru programatori C.
- Programarea de sistem se face în C. Dacă există sisteme care permit programarea de sistem în alte limbaje, aproape de fiecare dată acestea se bazează pe un interpretor care traduce celelalte limbaje în C. Sau un compilator specializat care a fost scris în C.
- Programele scrise în C depășesc în performanțe programele scrise în multe alte limbaje.

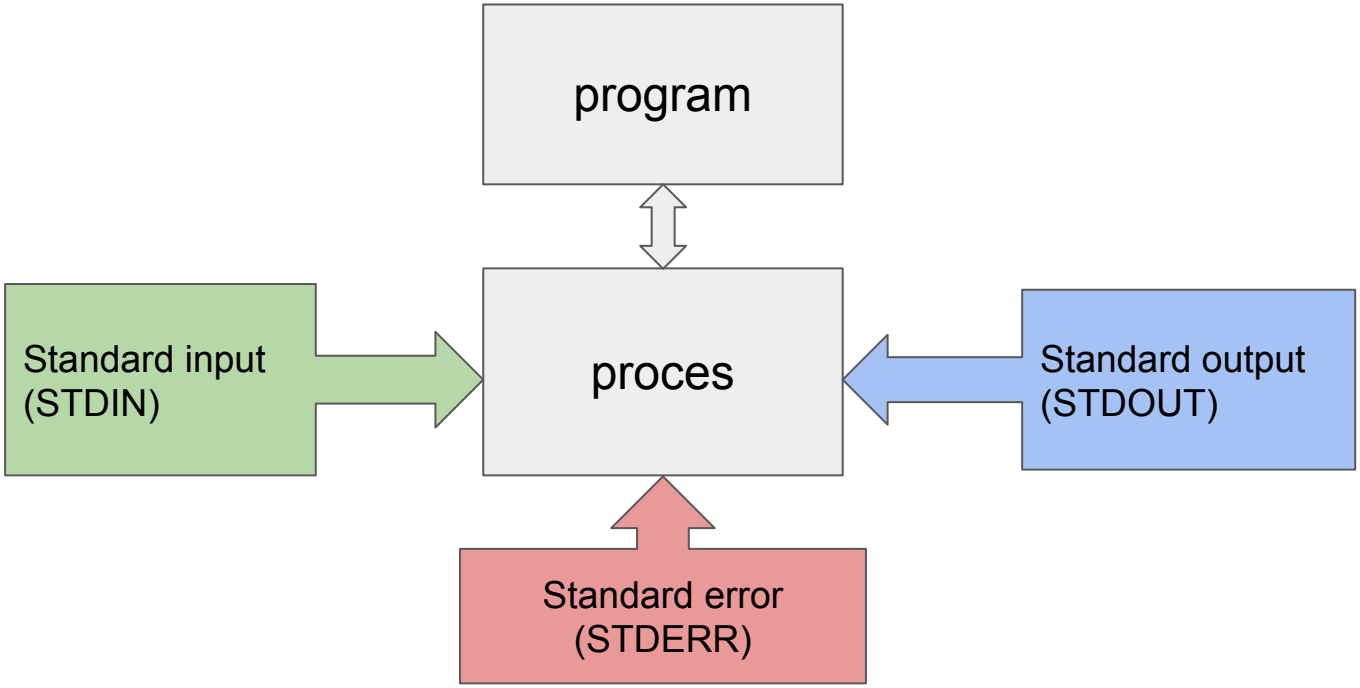
De ce să începem cu C?

- Sintaxa C-ului a stat la baza multor altor limbaje de programare (Java, C++ C#, perl, php, javascript… etc.)
- La nivel introductiv, nu se pot studia caracteristicile avansate ale altor limbaje (C#, C++, Java).
- Fiind un limbaj de nivel mediu, este un instrument didactic bun pentru dezvoltarea unei gândirii corecte.

Input / Output în cazul programelor care rulează în consolă

Standard Input, Standard Output și Standard Error sunt trei sisteme de comunicare în mod text care sunt atașate de către nucleul sistemului de operare fiecărui proces care rulează într-o consolă. Aceste trei sisteme de comunicare sunt văzute de fiecare proces ca niște fișiere speciale, din care se poate citi și în care poate scrie. Bibliotecile de sistem care însoțesc compilatoarele au instrucțiuni speciale pentru comunicarea cu aceste pseudo-fișiere.

Pentru că majoritatea programelor trebuie cel puțin să afișeze rezultate, majoritatea programelor pe care le veți vedea de aici înainte vor include o librărie de funcții care permite interacțiunea cu aceste puncte de comunicare cu sistemul de operare, numită stdio (STanDard Input/Output). Implementările funcțiilor variază de la un sistem de operare la altul dar modul de apelare al acestor funcții este uniform.



Hello World

```c
#include <stdio.h>
int main()
{
  printf("Hello World");
  return 0;
}
```

În mod tradițional, primul exemplu într-un limbaj de programare este un program care tipărește în spațiul de ieșire expresia “Hello World”.

Tradiția a început în cartea **A Tutorial Introduction to the Language B** scrisă de Brian Kernighan, cel care urma să scrie și prima carte de programare în C, împreună cu unul dintre creatorii C, Dennis Ritchie

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

Limbajul C nu conține instrucțiuni de intrare sau ieșire în nucleul său, operațiile respective sunt implementare în biblioteci dependente de sistem care însoțesc compilatoarele.

```c
int main()
{
  return 0;
}
```

Programele C sunt obligatoriu împărțite în funcții. Orice program trebuie să aibă cel puțin o funcție, obligatoriu numită **main**, care se execută prima automat. Cele două paranteze care însoțesc funcția main reprezintă un spațiu pentru argumente

```c
printf("Hello World");
```

printf este una dintre funcțiile definite în biblioteca stdio.h și este utilizată pentru imprimarea variabilelor la standard output.

```c
int main(int argc, char **argv)
{
   printf("Argc, %d\n", argc);
   printf("Argv, %c\n", argv[0]);
   printf("Argv, %c\n", argv[1]);
   return 0;
}
```

https://onlinegdb.com/B1r_XYEam

hello_args.c

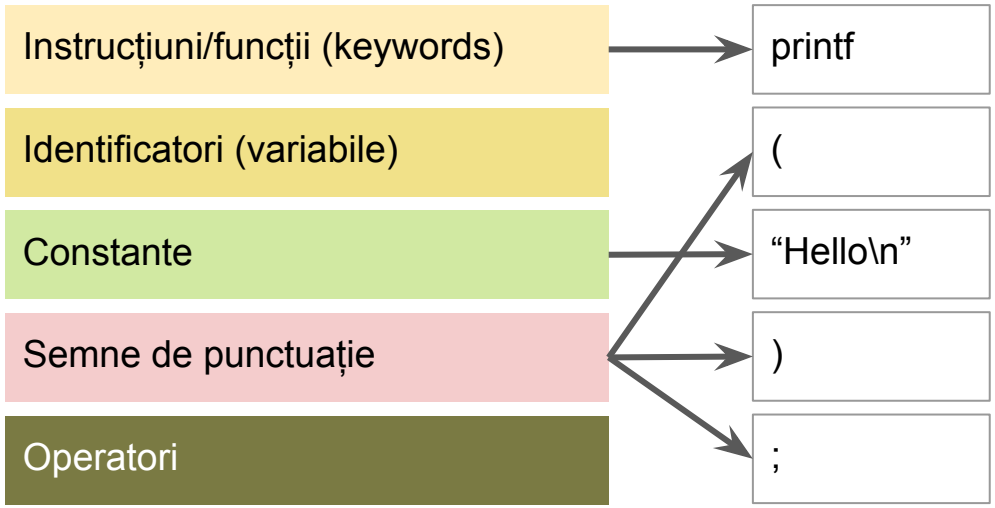
```
> ./hello_args ala
Numar argumente, 2
Argumentul 0, ./hello_args
Argumentul 1, ala
```

Declarația completă a funcției main poate adăuga două argumente: numărul de argumente și un șir de argumente care au fost date programului la rulare.

Elemente de sintaxă

Orice program C este văzut de compilator ca fiind format din fragmente (tokens) care pot fi de 5 feluri.

```
printf("Hello\n");
```




```c
#include <stdio.h>

int main()
{
  int N = 3;
  int a = 2;

  if (N >= 10)
   {
      a = 100;
      printf("N >= cu 10\n");
   }
  else if (N < 10 && N >= 0)
   {
      printf("N <= 10\n");
   }
  else {
      printf("N negativ\n");
  }
  printf("a dupa if %d\n", a);
  return 0;
}
```

if_elseif_else.c

https://onlinegdb.com/rktVhLrOv

- Declarațiile secvențiale se termină cu ;
- Instrucțiunile de preprocesor nu se termină cu ;
- Declarațiile de selecție, iterație nu se termina cu ;
- Apelurile de funcții nu se termină cu ;
- Comentariile sunt înconjurate de /* */
- Liniile goale sunt ignorate de preprocesor.
- Blocurile de cod sunt marcate de { }

Instrucțiuni C

auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if

Instrucțiunile reprezintă comenzile pe care le dăm compilatorului prin intermediul limbajului C. Împreună cu operatorii (ex: -, +, *, etc.) acestea (in număr de 32) reprezintă întregul conținut al limbajului C. Restul cuvintelor folosite în programele C sunt funcții scrise cu ajutorul acestor instrucțiuni si plasate în biblioteci (ex: stdio.h).

break, **case, etc.** - instructiuni de ciclare și selecție

char, **const, etc.** - instructiuni de declarare a variabilelor

auto, **extern, etc.** - instructiuni de declarare a variabilelor

enum, struct etc. - instructiuni de declarare a variabilelor

volatile - evită optimizarea variabilei de către compilator

sizeof - returnează dimensiunea unei variabile

int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while

UNDE E PRINTF ? Printf nu este o instrucțiune C. Limbajul C nu conține instrucțiuni IO. printf este parte din biblioteca de sistem, headerul stdio.h.

stdio.h

```
void     clearerr(FILE *);
char    *ctermid(char *);
char    *cuserid(char *);(LEGACY)
int      fclose(FILE *);
FILE    *fdopen(int, const char *);
int      feof(FILE *);
int      ferror(FILE *);
int      fflush(FILE *);
int      fgetc(FILE *);
int      fgetpos(FILE *, fpos_t *);
char    *fgets(char *, int, FILE *);
int      fileno(FILE *);
void     flockfile(FILE *);
FILE    *fopen(const char *, const char *);
int      fprintf(FILE *, const char *, ...);
int      fputc(int, FILE *);
int      fputs(const char *, FILE *);
size_t   fread(void *, size_t, size_t, FILE *);
FILE    *freopen(const char *, const char *, FILE *);
int      fscanf(FILE *, const char *, ...);
int      fseek(FILE *, long int, int);
int      fseeko(FILE *, off_t, int);
int      fsetpos(FILE *, const fpos_t *);
long int ftell(FILE *);
off_t    ftello(FILE *);
int      ftrylockfile(FILE *);
void     funlockfile(FILE *);
size_t   fwrite(const void *, size_t, size_t, FILE *);
int      getc(FILE *);
int      getchar(void);
int      getc_unlocked(FILE *);
int      getchar_unlocked(void);
```

```
int      getopt(int, char * const[], const char);
(LEGACY)
char    *gets(char *);
int      getw(FILE *);
int      pclose(FILE *);
void     perror(const char *);
FILE    *popen(const char *, const char *);
int      printf(const char *, ...);
int      putc(int, FILE *);
int      putchar(int);
int      putc_unlocked(int, FILE *);
int      putchar_unlocked(int);
int      puts(const char *);
int      putw(int, FILE *);
int      remove(const char *);
int      rename(const char *, const char *);
void     rewind(FILE *);
int      scanf(const char *, ...);
void     setbuf(FILE *, char *);
int      setvbuf(FILE *, char *, int, size_t);
int      snprintf(char *, size_t, const char *, ...);
int      sprintf(char *, const char *, ...);
int      sscanf(const char *, const char *, int ...);
char    *tempnam(const char *, const char *);
FILE    *tmpfile(void);
char    *tmpnam(char *);
int      ungetc(int, FILE *);
int      vfprintf(FILE *, const char *, va_list);
int      vprintf(const char *, va_list);
int      vsnprintf(char *, size_t, const char *, va_list);
int      vsprintf(char *, const char *, va_list);
```

Standarde C

1978 K&R C

- limbajul C

1990 C89 (ANSI C)

- prototipuri funcționale
- const, volatile, void, long float, long double
- pointeri către funcții fără dereferențiere

1999 C99

- funcții inline
- long long int, boolean, complex
- tablouri cu lungime variabila

2011 C11

- alinierea structurilor de date
- multi-threading
- renunțarea la funcția gets

2018 C18

- update tehnic, fara caracteristici noi

2021 C2X

Critici: C este un limbaj prea liber, lipsit de structura și control

- Nu există instrumente de control al execuției la run-time.
- Nu există instrumente de protecție a memoriei
- Nu există metode eficiente de management al erorilor.

The spirit of C (same as BCPL)

- trust the programmer
- don’t prevent the programmer from doing what needs to be done
- keep the language small and simple
- provide only one way to do an operation
- make it fast even if it is not guaranteed to be portable.

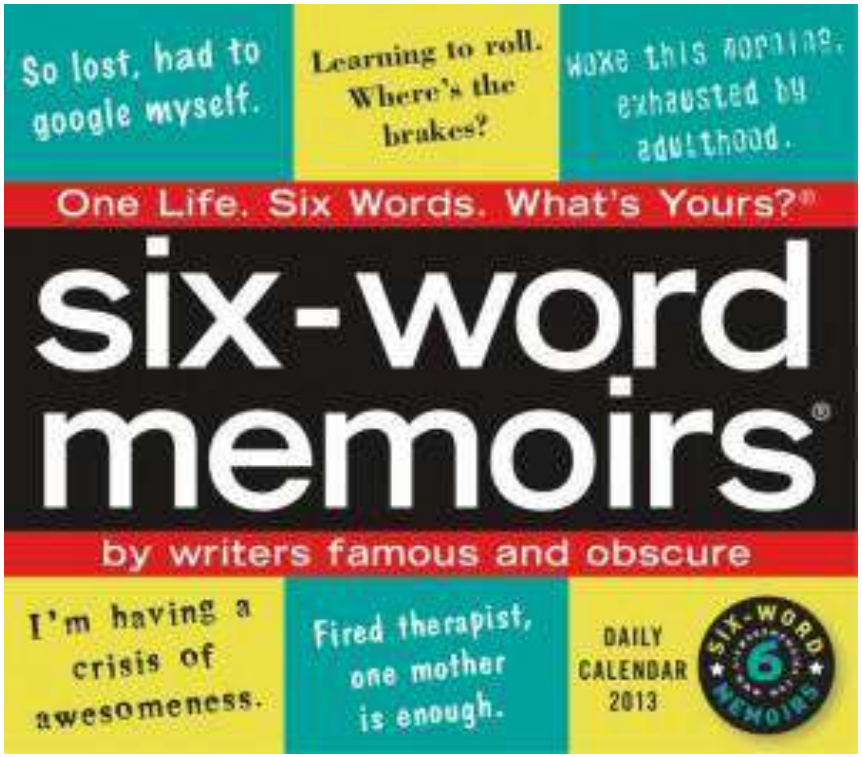
The philosophy of C is derived directly from BCPL.

C is the most precise representation of algorithms and programming details above assembly. In C **one can still recognize the relation with the hardware**. Upwords, things become much more abstract and conceptual.

Critici: C este un limbaj cu prea multe limitări

Alte limbaje oferă instrumente avansate pentru multe operații care în C trebuie implementate manual (ex: parcurgerea unui tablou fără cunoașterea dimensiunii acestuia cu instrucțiuni de tip foreach)

For sale: baby shoes, never worn - Ernest Hemingway



https://www.sixwordmemoirs.com/

Limitările nu limitează creativitatea (ci dimpotrivă)!!!

C - Variabile

Variabile

Programele operează cu date. Datele sunt de mai multe feluri: numele (întregi, zecimale), caractere (a,b,c… etc).

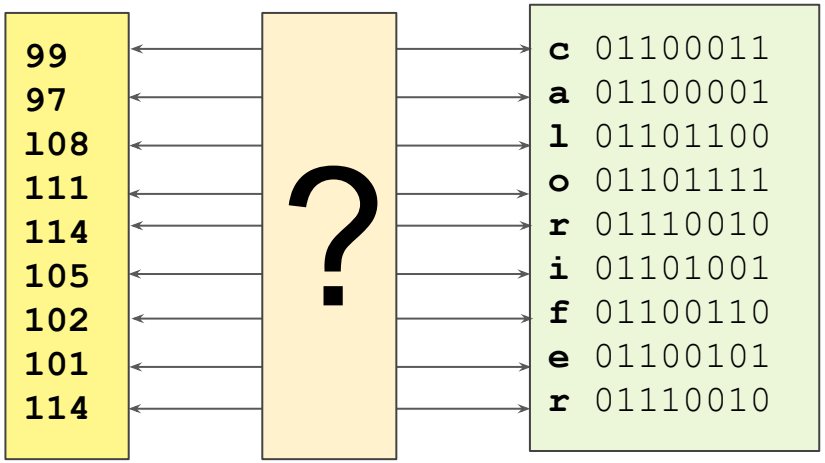
Pentru a opera cu acestea, ele trebuie să fie convertite în format binar și să ajungă în anumite spații de memorie

```
#include <stdio.h>

int main()
{
   int a = 20;
   int b = 30;
   printf("Suma este %i", a + b);
   return 0;
}
```

În memoria computerelor nu se pot stoca decât numere binare.

Pentru a putea stoca texte, e necesară o convenție care să permită stocarea lor sub formă de numere binare.

Compilatorul trebuie să știe însă despre ce e vorba.



Variabilele sunt etichete pentru colecții de adrese de memorie. Acestea se declară în interiorul programelor, adresele corespunzătoare lor sunt alocate în momentul compilării iar apoi programul operează doar cu locații de memorie

Numele variabilelor

Fiecare variabilă trebuie să aibă un nume. Numele acesteia **trebuie să fie unic** (nu pot exista două variabile cu același nume **în același scop**) și trebuie să respecte următoarele reguli:

- Un nume de variabilă poate conține numere și cifre, la care se adaugă semnul _ (underscore)
- Numele NU poate conține spații, virgule sau puncte.
- Un nume de variabilă NU poate începe cu o cifră
- Numele variabilelor nu poate fi aceleași cu cuvintele rezervate ale limbajului (nume de funcții, tipuri de variabile)
- Denumirile variabilelor și restricțiile trebuie să țină seama de faptul că limbajul C este sensibil la capitalizare (case sensitive).

Prin scopul variabilei se înțelege vizibilitatea ei de către alte blocuri de cod. Variabilele sunt:

- **locale** - vizibile doar în blocul de cod (funcție, buclă, bloc decizional) unde au fost definite
- **globale** - vizibile de toate elementele programului.

nume de variabile corecte

```
int suma = 0;
int eu = 12;
char cine_stie_ce[] = "da";
char la_revedere[] = "eh";
int aut112 = 0;
double fprint12 = 1.123;
float frintf_32 = 5.32;
```

nume de variabile incorecte

```
int su ma = 0;
int 12eu = 12;
char cine/stie_ce[] = "da";
char 1_la_revedere[] = "eh";
int aut.112 = 0;
double fprintf-12 = 1.123;
double fprintf = 5.32;
```

Atribuirea variabilelor

```
int x;
x = 2;
```

```
int x = 2;
```

```
int x = 2, y=5;
```

The $20 Million Bug

In Spring 1993, in the Operating System development group at SunSoft, we had a "priority one" bug report come in describing a problem in the asynchronous I/O library. The bug was **holding up the sale of $20 million worth of hardware** to a customer who specifically needed the library functionality, so we were extremely motivated to find it. After some intensive debugging sessions, the problem was finally traced to a statement that read :

```
x==2;
```

The statement as written **compared x to 2**, generated true or false, and discarded the result.

Peter van der Linden - Expert C Programming - deep C secrets

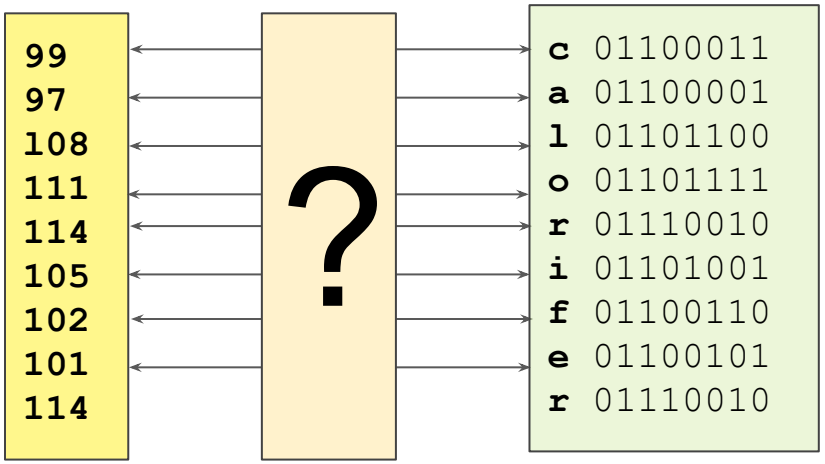
Tipul variabilelor - numere întregi

tip	dimensiune	minimum	maximum
char	1 octet	-127	127
unsigned char	1 octet	0	255
int	4 octeti	-2147483648	2147483648
unsigned int	4 octeti	0	4294967295
short	2 octeti	-32768	32768
unsigned short	2 octeti	0	65535
long	8 octeti	-9223372036854775808	9223372036854775808
unsigned long	8 octeti	0	18446744073709551615
long long	8 octeti	-9223372036854775808	9223372036854775808
unsigned long long	8 octeti	0	18446744073709551615

Tipul char are și altă utilizare

C nu are nici un tip special de date pentru stocarea textelor. Acestea se stochează cu ajutorul tipului **char** (număr întreg pe 8 biți) și a unei convenții cunoscută sub numele **ASCII** (American Standard Code for Information Exchange).

Codul ASCII asociază câte un caracter fiecărui număr întreg care se poate stoca în 8 biți.



Chiar dacă tipul **char** este utilizat pentru șiruri de caractere, acestea pot fi stocate (ineficient) în orice tip de variabila pentru numere întregi.

char_ex.c

```
#include <stdio.h>

int main()
{
    char v = 89;
    char f = 'a';
    printf("v intreg: %i\n", v);
    printf("v caracter: %c\n", v);
    printf("f intreg: %i\n", f);
    printf("f caracter: %c\n", f);

    return 0;
}
```

https://onlinegdb.com/SyuykO46Q

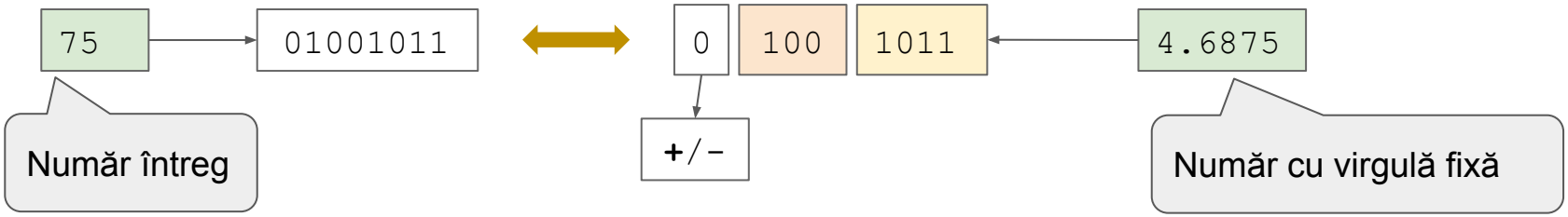
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^

Fragment din tabela ASCII.
https://commons.wikimedia.org/wiki/File:ASCII-Table.svg

Stocarea numerelor raționale / reale

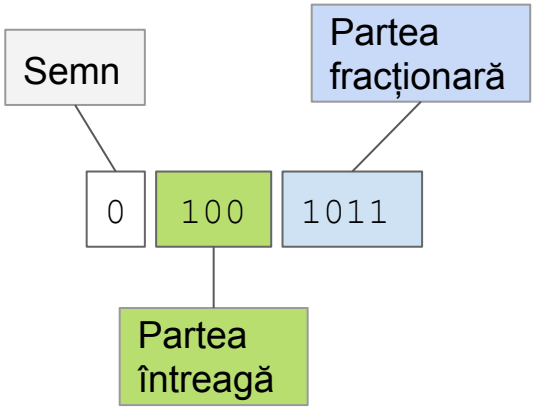
Stocarea cu virgulă fixă - exemplu în 8 biți

O posibilă variantă este să împărțim registrul în două părți (prin convenție) astfel încât o parte să stocheze numărul dinainte de virgulă și cealaltă parte cel de după virgulă



0	1	0	0	1	0	1	1		
	(1×2^{2})	(0×2^{1})	(0×2^{0})	(1×2^{-1})	(0×2^{-2})	(1×2^{-3})	(1×2^{-4})	=	4.6875

Câte numere raționale cu semn încap în 8 biți?



3 biti - 8 valori	
000.xxxx	0.xxxx
001.xxxx	1.xxxx
010.xxxx	2.xxxx
011.xxxx	3.xxxx
100.xxxx	4.xxxx
101.xxxx	5.xxxx
110.xxxx	6.xxxx
111.xxxx	7.xxxx
1000.xxxx	8.xxxx

128 de valori posibile, egal spațiate.

4 biti - 16 valori. Celelalte nu “încap”	
xxx.0000	x.0000
xxx.0001	x.0625
xxx.0010	x.1250
xxx.0011	x.1875
x.00110000000001101001	x.1876
x.0011000000001101001	x.1877
xxx.0100	x.2500
xxx.0101	x.3125
xxx.0110	x.3750
xxx.0111	x.4375
xxx.1000	x.5000
xxx.1001	x.5625
xxx.1010	x.6250
xxx.1011	x.6875
xxx.1100	x.7500
xxx.1101	x.8125
xxx.1110	x.8750
xxx.1111	x.9375

Ce contează mai mult: mărimea sau precizia?

Stocarea cu virgulă fixă are următoarele caracteristici:

- Numerele care pot fi stocate sunt **egal distanțate** iar distanța dintre ele depinde de numărul de biți ales pentru **stocarea zecimalelor**.
- Numerele maxime și minime care pot fi stocate depind de numărul de biți ales pentru **stocarea părții întregi**.

			Minim	Maxim	Distanță
0	111	1111	-7.9375	7.9375	0.0625
0	1111	111	-15.8750	15.8750	0.125
0	11111	11	-31.7500	31.7500	0.25

Un exemplu mai apropiat de realitate - stocarea numerelor în virgulă fixă în 32 de biți, folosind 16 biți pentru partea întreagă si 16 biti pentru zecimale. Numărul maxim care se poate stoca este:

1111111111111111.1111111111111111 = 65535.9999847412109375

C **nu suportă** direct calculele cu virgulă fixă. Anumite compilatoare au implementate extensii pentru aceasta. Se pot face calcule în virgulă fixă în C și fără aceste extensii cu utilizarea operatorilor de bit-shifting.

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbcpx01/decchap.htm

Stocarea cu virgulă mobilă - exemplu în 8 biți

Orice număr real poate fi scris în mai multe feluri:

$$3456 = 34.56 * 100 = 34.56 * 10^2 = 3.456 * 1000$$

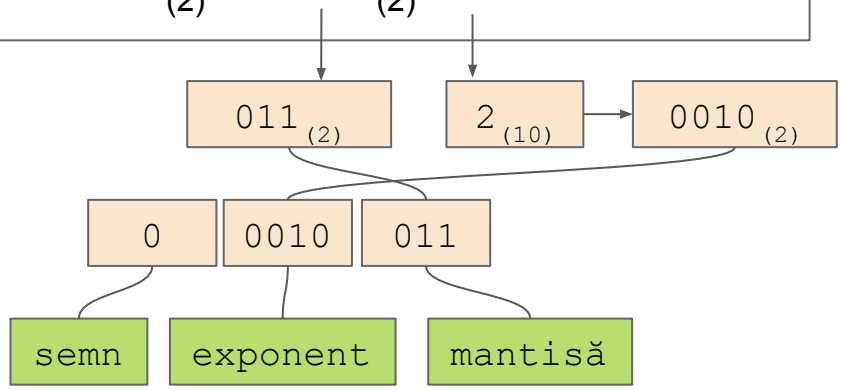
$$3.456 * 10^3$$

Modul în care se reține o singură cifră în stânga virgulei iar exponentul se calculează corespunzător se numește **notația normalizată**

Notația normalizată **se poate transfera în baza 2**. Astfel, dacă avem la dispoziție un număr oarecare de biți pentru stocarea unui număr, putem decide ca o parte din acei biți să fie utilizați pentru **stocarea zecimalelor** și ceilalți pentru **stocarea exponentului**.

Putem alege exponentul lui 2 astfel încât în stânga virgulei să fie mereu 1 (în baza 2). În acest caz, avem nevoie să stocăm doar cifrele din dreapta virgulei (mantisă) și exponentul

$$5.5_{(10)} = 101.1_{(2)} = 1.011_{(2)} \times 2^2$$

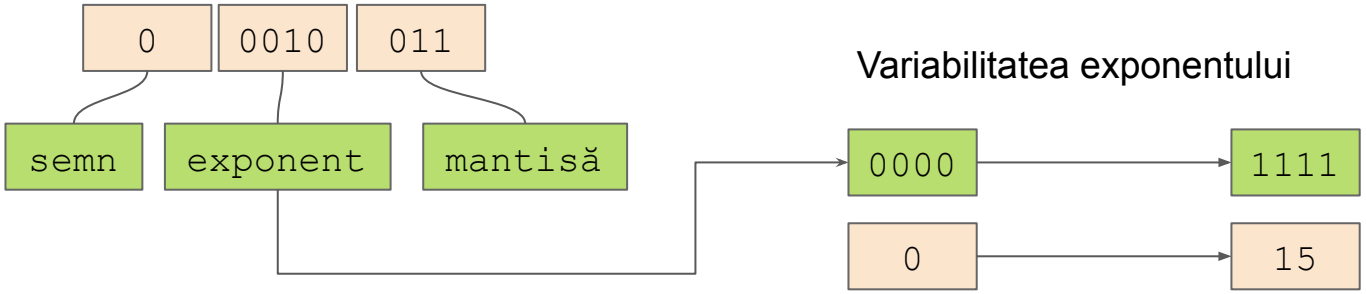


În realitate, stocarea numerelor flotante în mai puțin de 4 octeți nu este utilă pentru calculele generale.

Numerele flotante stocate în mai puțin de 4 biți se numesc **minifloats** și pot fi utilizate în anumite situații speciale.

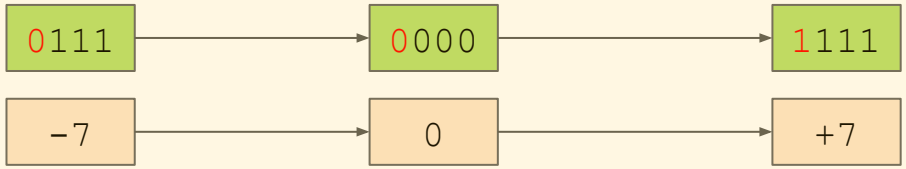
Mai multe detalii:
https://en.wikipedia.org/wiki/Minifloat

Stocarea cu virgulă mobilă - exemplu în 8 biți



Unde sunt exponenții negativi? În lipsa lor nu putem stoca numele subunitare

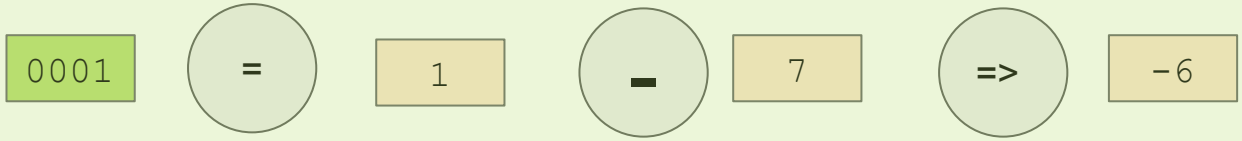
1: folosim unul dintre biții exponentului pentru a stoca semnul



NU

2: folosim "exponent bias" = 2k-1 unde k este numărul de biți din exponent

exponentul are 4 biți => k=4 => exponent bias = 2 * 4 -1 = 7



DA

8 biți virgulă mobilă - ce numere putem stoca?

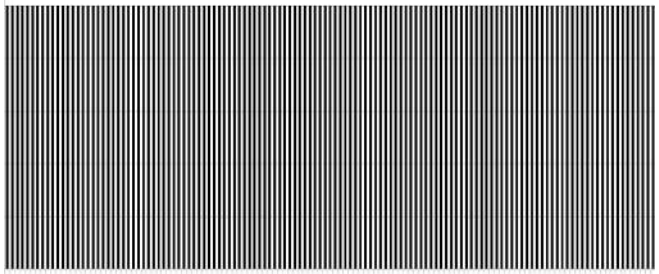
stocare	decimal	binar
00000001	0.125	0.001
00000010	0.250	0.00001
NU	0.31251	0.**0101**000000000000101
00000011	0.375	0.011
00000100	0.5	0.1
NU	0.51	0.**1000**001010001111011
00000101	0.625	0.101
NU	0.62520	0.1010000000001101001

01100000	2048	**1000**00000000
NU	2100	**1000**00110100
01100001	2304	**1001**00000000

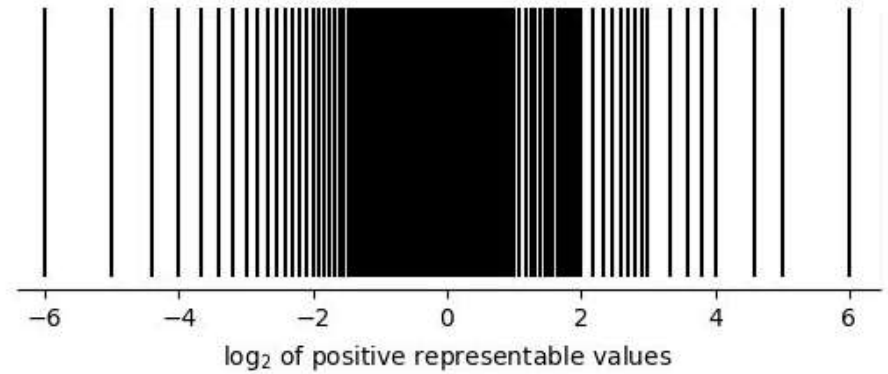
Câteva lucruri interesante despre numerele flotante

Sistemul de stocare a numerelor în virgulă mobilă e extrem de ingenios dar nu poate depăși limitările naturale ale stocării în format binar. Astfel, 8 biți pot stoca 256 de numere diferite și nu mai multe de atât, indiferent de modul în care le reprezentăm.

Densitatea numerelor cu virgulă fixă pozitive stocate in 8 biti.



Densitatea numerelor flotante stocate in 8 biti.



Cu cat numărul de biți în care sunt stocate numerele flotante **crește**, cu atât **crește și precizia acestora** - numerele stocate sunt mai apropiate de numerele introduse. Dar, numerele stocate în acest mod nu sunt niciodată precise până la capăt.

Distanța dintre numerele care pot fi stocate ca float-uri **crește exponențial** cu cât numărul se îndepărtează (spre plus sau minus) de zero.

Mult mai multe informații: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Stocarea numerelor în variabile flotante

0.7	semn		exponent		mantisa		total
float	0	1b	01111110	8b	01100110011001100110011	23b	32 biți
double	0	1b	01111111110	11b	0110011001100110011001100110011001100110011001100110	52b	64 biți

float	0.69999998807907104492187 5
double	0.69999999999999999555910790149 9

Valorile stocate în realitate

1.1	semn		exponent		mantisa		total
float	0	1b	0111111	8b	10001100 11001100 11001101	23b	32 biți
double	0	1b	01111111111	11b	0001100110011001100110011001100110011001100110011010	52b	64 biți

float	1.100000023841857910156 25
double	1.100000000000000088817841970 01

Valorile stocate în realitate

https://www.exploringbinary.com/floating-point-converter/

https://www.binaryconvert.com

Sistemul de stocare al numerelor flotante este un sistem de compresie aproximativa.

Nu se poate stoca orice număr, indiferent de dimensiunea în biți.

Operațiile aritmetice NU vor avea tot timpul rezultatul corect.

Operatori ca =, <, > pot returna rezultate greșite!!

Tipuri de date în virgulă mobilă disponibile în C

tip	dim	interval	cea mai mică valoare pozitivă	nr. zecimale
float	4 octeti	±3.4E+38	1.2E-38	6
double	8 octeti	±1.7E+308	2.3E-308	15
long double	10 octeti	±1.1E+4932	3.4E-4932	19

printf convertește automat float-ul la double. Evident, cantitatea de date stocate în float nu se modifică dar această conversie ne permite să tratăm floatul ca double pentru a vedea precizia.

```
-------------FLOATS------------
a este: 1.111111164093018
b este: 1111.111083984375000
c este: 11111111.000000000000000
d este: 111111110656.000000000000000
e este: 111111114326016.000000000000000
-------------DOUBLES------------
m este: 1.111111111111111
n este: 1111.111111111111086
o este: 11111111.111111111938953
p este: 111111111111.111114501953125
q este: 111111111111111.109375000000000
```

floats_ex.c

```c
#include <stdio.h>

int main() {

  float a = 1.111111111111111111111111;
  float b = 1111.111111111111111111111;
  float c = 11111111.111111111111111111;
  float d = 111111111111.111111111111111;
  float e = 111111111111111.1111111111;

   printf ("-------------FLOATS------------\n");
   printf ("a este: %.15f\n", a);
   printf ("b este: %.15f\n", b);
   printf ("c este: %.15f\n", c);
   printf ("d este: %.15f\n", d);
   printf ("e este: %.15f\n", e);

  double m = 1.111111111111111111111111;
  double n = 1111.111111111111111111111;
  double o = 11111111.111111111111111111;
  double p = 111111111111.111111111111111;
  double q = 111111111111111.1111111111;

   printf ("-------------DOUBLES------------\n");
   printf ("m este: %.15lf\n", m);
   printf ("n este: %.15lf\n", n);
   printf ("o este: %.15lf\n", o);
   printf ("p este: %.15lf\n", p);
   printf ("q este: %.15lf\n", q);

  return 0;
}
```

https://onlinegdb.com/rk5nMuSuw

Dacă mărimea nu contează

Tipuri de date întregi	
char	1 octet
unsigned char	1 octet
int	4 octeti
unsigned int	4 octeti
short	2 octeti
unsigned short	2 octeti
long	8 octeti
unsigned long	8 octeti
long long	8 octeti
unsigned long long	8 octeti

Caractere	Char
Siruri de caractere	Char[]

C are doar două tipuri de date: numere întregi și flotante.

Numere flotante

Numere intregi

tipuri de date flotante	
float	4 octeti
double	8 octeti
long double	10 octeti

Toate tipurile de date fundamentale în C au **o corespondență directă în hardware**. Nu există tipuri de bază pentru numere complexe (ca în FORTRAN). Compilatorul nu trebuie să facă eforturi de a stoca tipuri complicate de date.

C a început să permită float-uri odată cu apariția suportului hardware.

Tipurile de date sunt pentru compilator

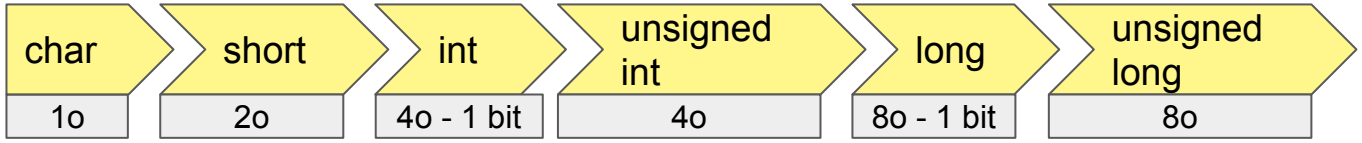
char	K	⇨	01001011
fixed	4.6875	⇨	01001011
int	75	⇨	01001011
float	5.5	⇨	01001011

Instrucțiunile care ajung la procesor sunt “scrise” de compilator astfel încât să opereze corect cu datele aflate în memorie. Dacă se știe că 01001011 este un char, atunci el va fi afișat ca atare. Dacă este float, se vor aplica operațiile corespunzătoare numerelor în virgulă mobilă.

Compilatorul trebuie să știe ce tip de date vrem să punem în variabile. Uneori poate să ghicească

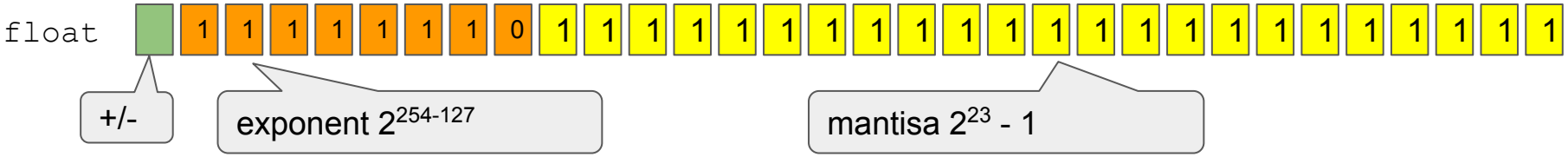
Conversii de tip automate

variabilele sunt convertite în tipuri cu spațiu de stocare mai mare sau egal



Compilatoarele permit și conversii de tipul:

Cum putem trece de la 8 octeți la 4 octeți?



Cel mai mare număr întreg care poate fi stocat într-un float (32):

$2^{254-127} \cdot (1 + (2^{23}-1)/2^{23}) = 2^{127} \cdot (2-2^{-23}) = 2^{128}-2^{104}$

=

340282346638528859811704183484516925440

Cel mai mare număr întreg (unsigned long long):

18446744073709551615

modul de stocare a numerelor flotante permite acceptarea unor numere întregi mai mari decât cel mai mare unsigned long long.

conversia unsigned long long -> double este imprecisă

18446744073709546496
18446744073709548544

18446744073709547520

semn: 0
exp: 10000111110
mantisa: 11111111111111111111111111
11111111111111111111111110

18446744073709548545
18446744073709550591

18446744073709549568

semn: 0
exp: 10000111110
mantisa: 11111111111111111111111111
11111111111111111111111111

18446744073709550592
18446744073709551615
18446744073709553664

18446744073709551616

semn: 0
exp: 10000111111
mantisa: 00000000000000000000000000
00000000000000000000000000

18446744073709553665
18446744073709557759

18446744073709555712

semn: 0
exp: 10000111111
mantisa: 00000000000000000000000000
00000000000000000000000001

Pentru operații precise cu numere mari, se folosesc biblioteci auxiliare. Exemplu, https://gmplib.org/

Conversii de tip explicite

explicit_cast_1.c

```c
#include <stdio.h>

int main() {

  int a = 17, b = 5;
  double c;

  c = a / b;

  printf("Rezultatul este: : %f\n", c );

  c = (double) a / b;

  printf("Rezultatul este : %f\n", c );

  return 0;
}
```

https://onlinegdb.com/BJy1mjDvP

```
Rezultatul este: 3.000000
Rezultatul este:: 3.400000
```

sum este int, count este int. În mod normal, operațiile cu numere întregi produc numere întregi.

Împărțirea a două numere întregi va produce partea întreagă a rezultatului real. 17/5 = 3.4, partea întreagă este 3. Această valoare ajunge în variabila c

c este float, așa că va fi afișat cu zecimale

(double) pus în fața operației determină conversia rezultatului la tipul double. Astfel, în variabila c va ajunge rezultatul corect.

Această operație se numește conversie de tip explicită (type casting).

Conversii de tip explicite

```c
#include <stdio.h>

int main()
{
    float a = 1.1;
    float b = 2.2;

    float c = a + b;

    float d = (int) (a + b);

    float e = (int) a + b;

    printf ("c este: %f\n", c);
    printf ("d este: %f\n", d);
    printf ("e este: %f\n", e);
    return 0;
}
```

explicit_int_conversion.c

https://onlinegdb.com/B12omoruP

Operația adună două numere flotante și pune suma în cel de-al treilea număr flotant. Rezultatul este corect, 3.3.

Operatia de cast afectează rezultatul operației a + b (datorită parantezelor). Rezultatul final va fi un număr întreg.

Operația de cast afectează doar variabila a pe care o transformă în număr întreg (1). Rezultatul va fi 1 + 2.2 adică 3.2.

```
c este: 3.300000
d este: 3.000000
e este: 3.200000
```

Conversii de tip explicite

explicit_cast_2.c

```c
#include <stdio.h>

int main() {

  float a = 3.12345;
  float b = 7.72213;
  float c = 9.51434;

  printf("suma este : %f\n", (a + b + c) );
  printf("suma este : %i\n", (int)(a + b + c) );
  printf("suma este : %i\n", (int)a + (int)b + (int)c );

  return 0;
}
```

https://onlinegdb.com/Bk5oSuE6X

```
suma este : 20.359921
suma este : 20
suma este : 19
```

trunchierea este făcută după adunare

trunchierea este făcută înainte de adunare

explicit_cast_3.c

```c
int main()
{
   long aa = 999999999999999999;
   float bb = 1.1417654;
   printf("aa - long - este %li \n", aa);
   printf("aa - int -  este %i \n", (int)aa);
   printf("aa - short - este %i \n", (short)aa);

   printf("bb - float - este %f \n", bb);
   printf("bb - int - este %i \n", (int)bb);
   printf("bb - int as float - este %f \n", (int)bb);
   return 0;
}
```

https://onlinegdb.com/H1VN8OEa7

```
aa - long - este
999999999999999999
aa - int -  este -1486618625
aa - short - este -1
bb - float - este 1.141765
bb - int - este 1
bb - int as float - este
1.141765
```

Constante

Constantele sunt nume date unor valori care nu pot fi modificate în timpul rulării programului. Pentru declararea lor se folosește cuvântul “const”, urmat de tip. Constantele **nu sunt stocate în stack**, declararea lor (când are sens) poate elibera o parte din stack.

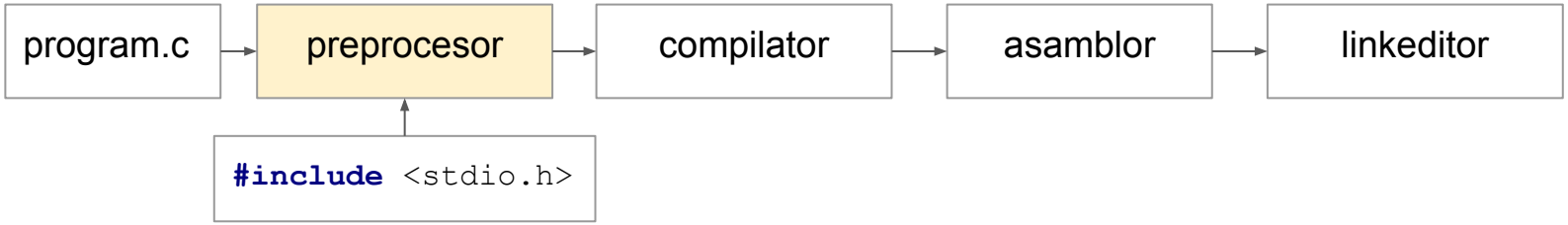
const.c

```c
#include <stdio.h>

int main() {

 const int a = 20;
 const int b = 10;
 const int c = 15;
 const int d = 5;
 int e;

 e = a + b * c / d;       // 20 +  15
 printf("a este : %i\n",  a );
 printf("b este : %i\n",  b );
 printf("c este : %i\n",  c );
 printf("d este : %i\n",  d );
 printf("Valoarea a + b * c / d este : %d\n",  e );

  return 0;
}
```

```
a este : 20
b este : 10
c este : 15
d este : 5
Valoarea a + b * c / d este : 50
```

https://onlinegdb.com/rkJwRF7hQ

#define



Pe lângă #include, preprocesorul oferă macro-ul #define care înlocuiește înainte de compilare toate aparițiile unei anumite etichete cu altceva.

```
#define PI 3.14107
```

Avantajul utilizării #define: constantele **NU mai ocupă loc suplimentar** în memorie.

const_define.c

```
#include <stdio.h>

int main() {
 #define a 20
 #define b 10
 #define c 15
 #define d 5
 int e;
 e = a + b * c / d;        // 20 +  15
 printf("a este : %i\n",  a );
 printf("b este : %i\n",  b );
 printf("c este : %i\n",  c );
 printf("d este : %i\n",  d );
 printf("Valoarea a + b * c / d este : %d\n",  e );
 return 0;
}
```

```
a este : 20
b este : 10
c este : 15
d este : 5
Valoarea a + b * c / d este : 50
```

https://onlinegdb.com/HyrOoYm2m

Clasa de stocare

O definire completă a unei variabile implică specificarea clasei de stocare. Ea nu este necesară de cele mai multe ori, **în lipsa precizării** acesteia se va considera că variabila este definită ca **auto**.

Clasa de stocare determină vizibilitatea unei variabile precum și segmentul de memorie unde este alocată.

Clasa	Segmentul de memorie în care este stocată variabila	Valoarea inițială după declarare	Vizibilitate a variabilei (scop)	Dealocare (până când este disponibilă)
auto	stack	garbage	in blocul de cod	sfârșitul blocului de cod
extern	data	zero	global, mai multe fișiere	sfârșitul programului
static	data	zero	in blocul de cod	sfârșitul programului
register	registrul CPU	garbage	in blocul de cod	sfârșitul blocului de cod

Variabile în alte limbaje de programare

O variabilă definită cu un anumit tip în limbajul C va rămâne de acel tip până la terminarea programului. Conversia de tip nu o modifică - conversia de tip “extrage” conținutul acelei variabile și îl interpretează conform noului tip dar variabila rămâne în memorie așa cum a fost definită inițial.
De asemenea, fiecare variabilă în C **trebuie definită împreună cu tipul acesteia**, de fiecare dată

Limbajele care au acest comportament se numesc **statice** (statically typed).

Un astfel de program nu se compilează dacă se încearcă atribuirea unei valori diferite, care nu se poate rezolva prin casting implicit.

```
#include <stdio.h>

int main()
{
   double a;
   a = 13;
   a = "Text";
   return 0;
}
```

```
static_typed.c:7:9: error:
incompatible types when
assigning to type ‘double’ from
type ‘char *’
```

Există limbaje de programare care nu se comportă astfel. Php nu are nici o problemă cu reatribuirea unui tip cu totul diferit de date.

Limbajele care se comportă astfel se numesc limbaje **dinamice** (dynamically typed), în general limbaje interpretate (care nu se transformă în executabile ci sunt executate de către un program special, numit interpretor).

```
<?php

$a = 10;
echo $a."\n";
$a = "Text";
echo $a ."\n";
```

```
10
Text
```

Stocarea variabilelor în Php

php este un limbaj interpretat - există un program special, numit **interpretor**, care “citește” codul php si îl transformă. Interpretorul php este scris în C. Interpretorul NU transformă codul php în cod C, ci apelează funcțiile nucleului și ale bibliotecilor de sistem pentru a executa fiecare instrucțiune în parte.

Interpretarea codului php presupune stocarea în memorie a variabilelor declarate în acesta, stocarea se face cu ajutorul unor structuri C. Datorită acestui mod de stocare (care nu implică alocare directă a memoriei) este posibilă modificarea tipului de date stocat într-o variabilă în timpul rulării programului (cu prețul performanței, evident).

```
typedef struct  zval struct {
   zvalue value value;          /* variable value */
   zend uint refcount  gc;      /* reference counter */
   zend uchar type;             /* value type */
   zend_uchar is_ref__gc;       /* reference flag */
} zval;
```

Orice variabilă definită în php se transformă într-o structură de tip zval. în timpul rulării, aceasta poate fi alterată.

```
typedef union  zvalue value {
   long lval;                   /* long value */
   double dval;                 /* double value */
   struct {
       char *val;
       int len;                 /* this will always be set for strings */
   } str;                       /* string (always has length) */
   HashTable *ht;               /* an array */
   zend object value obj;       /* stores an object store handle, and handlers */
} zvalue_value;
```


https://www.php.net/manual/ro/internals2.variables.intro.php

Tipizare slabă sau puternică

Un alt mod de analiza a sistemului de tipuri este legata de cat de obsesiv sunt “păzite” tipurile variabilelor.

Paznicul variabilelor este compilatorul - caracterizarea tipizării se referă exclusiv la acesta, nu neapărat la limbaj în sine.

Static/dinamic reprezinta “legea”, **slab/puternic** reprezinta intensitatea aplicarii ei.

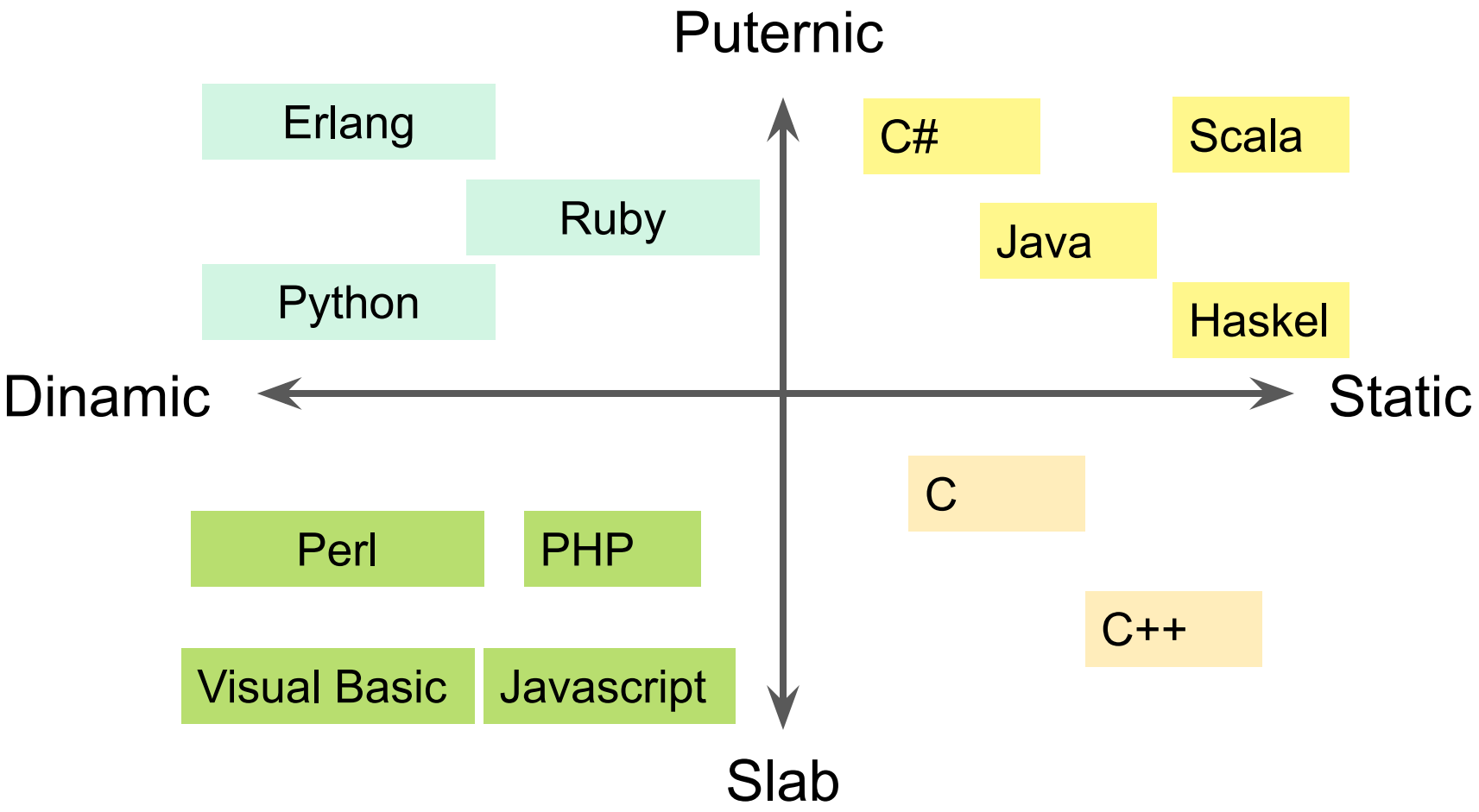
C este liberal, compilatorul încearcă tot ce poate pentru a face presupuneri cu privire la tipurile variabilelor. Atunci când poate, le convertește automat. Aceasta este o **tipizare slabă**.

Expunerea pointerilor către programator este de asemenea considerată o caracteristică a slăbiciunii C-ului din punct de vedere al tipizării.

Limbajul de asamblare este netipizat - nu există nici o verificare a tipurilor variabilelor, aceasta este lăsată complet în grija programatorului.

Tipizarea în diferite limbaje de programare

Dinamic: se poate schimba tipul de date stocat într-o variabilă în timpul rulării programului
Static: Tipul datelor stocate într-o variabilă nu poate fi modifcat in timpul rulării
Slab: Se pot face transferuri între tipuri diferite de date acolo unde compilatorul consideră posibil.
Puternic: Nu se pot face transferuri implicite între tipuri diferite de date



Conversia implicită de tip

Explozia rachetei Arianne 5 (4 iunie, 1996)

The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.*

https://itsfoss.com/a-floating-point-error-that-caused-a-damage-worth-half-a-billion/

Racheta Patriot, 25 Februarie, 1991

Racheta Patriot nu a reușit să intercepteze o rachetă SCUD lansată de forțele irakiene. Racheta SCUD a lovit o bază americană, omorînd 20 de oameni

Because of the way the Patriot computer performs its calculations and the fact that its registers are only 24 bits long, the conversion of time from an integer to a real number cannot be any more precise than 24 bits. ... Consequently, performing the conversion after the Patriot has been running continuously for extended periods causes the range gate to shift away from the center of the target, making it less likely that the target, in this case a Scud, will be successfully intercepted

http://www-users.math.umn.edu/~arnold//disasters/patriot.html

Typedef

typedef se poate folosi pentru a redenumi un anumit tip de variabila. Typedef NU crează un nou tip de variabilă ci mai degrabă crează o poreclă cu ajutorul căreia se pot defini mai departe variabilele.

```
typedef unsigned char  u8;
typedef signed char   s8;
typedef unsigned short u16;
typedef signed short   s16;
typedef unsigned long  u32;
typedef signed long   s32;
```

In cod:

```
unsigned char var;
```

se va înlocui cu:

```
u8 var;
```

```
typedef struct Books {
  char title[50];
  char author[50];
  int book_id;
} Book;
```

In cod, declararea structurii se va face:

```
Book book;
```

Please don't introduce more typedefs. They only hide what the hell the thing is, which is actively _bad_ for structures, since passing a structure by value etc is something that should never be done, for example.

The few saved characters of typing do not actually _buy_ you anything else, and only obscures what the thing is.

Linus Torvalds

input/output

printf

int printf(const char *fmt, ...)

funcțiile de acest tip se numesc “variadice”

Declarația funcției ne arată:

1. Există un argument obligatoriu de tip char * (pointer către char adică un șir de caractere)
2. Acesta poate fi urmat de unul sau mai multe argumente.

Argumentul obligatoriu este un **șir de caractere** numit “format”. Șirul respectiv poate avea o serie de coduri speciale pentru afișarea variabilelor care urmează ca argumente suplimentare.

```
ex: printf(“număr întreg: %i si număr flotant %f”, 23, 23.4567);
```

scanf

int scanf(const char *fmt, ...)

funcțiile de acest tip se numesc “variadice”

Declarația funcției ne arată:

1. Există un argument obligatoriu de tip char * (pointer către char adică un șir de caractere)
2. Acesta poate fi urmat de unul sau mai multe argumente.

```
ex: scanf(“%i”, &intvar);
```

Pentru a putea plasa valoarea obținută de la consolă într-o variabilă definită în afara funcției scanf, trebuie să trimitem o **referință** către o variabilă **deja existentă**.

printf

%	flags	nr. minim caractere	.	precizie	lungimea tipului de date	specificator

Specificator	Descriere
i sau d	Număr întreg cu semn
u	Număr întreg fără semn
o	Număr octal fără semn
x sau X	Număr întreg în format hexazecimal fără semn
f sau F	Număr flotant
e sau E	Notație științifică (mantisă + exponent)
g sau G	Alege cea mai scurtă reprezentare între %f sau %e
a sau A	Număr flotant hexazecimal
c	Caracter
s	Șir de caractere
p	Adresa unui pointer
%	Semnul %

printf - lungimea tipului de date

%	flags	nr. minim caractere	.	precizie	lungimea tipului de date	specificator

	specificator			
lungime	d, i	u,o,x,X	f,F,g,g,e,E,a,A	c
-	int	uns. int	double	int
hh	signed char	uns. char		
h	short int	uns. short int		
l	long int	uns. long int		wint_t
ll	long long int	uns. long long int		
j	intmax_t	uintmax_t		
z	size_t	size_t		
t	ptrdiff_t	ptrdiff_t		
L			long double	

printf

%	flags	nr. minim caractere	.	precizie	lungimea tipului de date	specificator

flags	descriere
-	Aliniază la stânga
+	Adaugă semnul la numărul afișat, chiar dacă acesta e pozitiv
spațiu	Afișează semnul - sau spațiu
#	Adaugă virgula chiar dacă nu e necesară
0	Adaugă 0 în stânga numărului în loc de spațiu când e cazul.

nr. minim de caractere	descriere
număr	numărul minim de caractere afișate. Dacă valoarea este mai mare decât valoarea afișată se vor adăuga spații
*	numărul minim de caractere afișate se citește ca parametru

printf

%	flags	nr. minim caractere	.	precizie	lungimea tipului de date	specificator

precizie	descriere
număr	În cazul numerelor întregi, precizia specifică numărul de caractere scrise. Dacă numărul e mai mare, nu va fi trunchiat. Dacă este mai mic, se vor adăuga spații. Pentru numere flotante, precizia specifică numărul de zecimale afișat.
.*	numărul minim de caractere afișate se citește ca parametru

Șirurile de caractere imprimate cu printf pot conține o serie de coduri ajutătoare numite “escape codes”

\\	Adaugă caracterul \
\n	Adaugă un caracter de tip “newline”
\b	Adaugă un caracter de tip “backspace” (se mută înapoi un caracter).
\t	Adaugă un caracter de tip tab.

printf - exemple pentru numere întregi

printf_int.c

```c
#include <stdio.h>

int main()
{
   char a = 112;
   int b = 12345;
   long int c = 543432345245;

   //Numar intreg
   printf ("%d\n", a);
   // Caracter
   printf ("%c\n", a);
   //Numar intreg
   printf ("%d\n", b);
   //Numar intreg cu dimensiunea long double
   printf ("%ld\n", c);
   //Numar intreg cu 12 cifre
   printf ("%12d\n", b);
   //Numar intreg cu 12 cifre cu 0 adaugat
   printf ("%012d\n", b);
   //Numar intreg cu 10 cifre ca parametru
   printf ("%*d\n",10, b);
   return 0;
}
```

```
112
p
12345
543432345245
    12345
000000012345
    12345
```

https://onlinegdb.com/BkH3WDq27

printf - exemple pentru floats

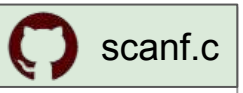
printf_floats.c

```c
#include <stdio.h>

int main()
{
   float c = 342.43543453;
   //Utilizand doar specificatorul %f
   printf("%f\n", c);
   //Specificatorul de float cu semnul fortat
   printf("%+f\n", c);
   //Specificatorul de float cu minus sau spatiu
   printf("% f\n", c);
   //Specificatorul de float cu numar si precizie
   printf("%5.2f\n", c);
   //Specificatorul de float cu partea intreaga si precizie
   printf("%7.2f\n", c);
   //Specificatorul de float cu partea intreaga si precizie ca parametrii
   printf("%*.*f\n", 8, 5, c);
   return 0;
}
```

https://onlinegdb.com/rk2WMPcn7

```
342.435425
+342.435425
 342.435425
342.44
 342.44
342.43542
```

scanf




```c
#include <stdio.h>

int main()
{
   int s;
   float b;
   char c;
   printf("Introduceti un numar intreg:");
   scanf ("%d", &s);
   printf ("Ati introdus numarul %d\n",s);
   printf("Introduceti un numar rational:");
   scanf ("%f", &b);
   printf ("Ati introdus numarul %f\n",b);
   printf("Introduceti un caracter:");
   scanf (" %c", &c);
   printf ("Ati introdus caracterul %c\n",c);
   return 0;
}
```

https://onlinegdb.com/ry_y_P92X

```
Introduceti un numar intreg:12
Ati introdus numarul 12
Introduceti un numar rational:32.45
Ati introdus numarul 32.450001
Introduceti un caracter:d
Ati introdus caracterul d
```

Operatorul & plasat înaintea numelui variabilei specifică o **referință** către acea variabilă.

Referința conține **adresa segmentului de memorie** în care este stocată variabila.

Astfel, scanf poate prelua datele introduse de la consolă și le poate **plasa direct în memorie** acolo unde restul programului știe că va găsi variabilele noastre.

Dacă am trimite variabila, în locul referinței, aceasta ar fi **copiată în altă adresă de memorie** la intrarea în funcție, adresă care ar fi **ștearsă** la părăsirea funcției iar informația s-ar pierde.

Funcțiile sunt tipizate în C, de aceea nu putem returna din scanf datele pentru că nu știm ce fel de date ne trebuie.

~~float cc = scanf(...)~~

ar însemna să avem un scanf pentru char, unul pentru int, unul pentru float… etc.

Comentarii

```
/* Comentariu in limbajul C */

/*

Comentariul poate fi pe mai multe linii
(oricate).

*/

#include <stdio.h>

// Majoritatea compilatoarelor accepta si comentariile precedate de doua
slashuri

int main()
{

   return 0;
}
```

Programele sunt scrise nu numai pentru compilator dar și pentru alți oameni. Alte persoane sau aceeași persoană câțiva ani mai târziu. Pentru aceștia, comentariile pot fi uneori mai importante decât codul.

De multe ori citirea codului altcuiva este mai lentă decât scrierea unui cod nou. Comentariile ajută.

Operatori

- Operatori aritmetici
- Operatori logici
- Operatori relaționali
- Operatori la nivel de bit
- Operatori de atribuire
- Alți operatori

Operatori aritmetici

```
int a = 10;
int b = 26;
```

Observați problemele de precizie!!

Op	Descriere	**int** a = 10; **int** b = 26;	**float** a = 10.12; **float** b = 26.32;	**double** a = 10.12; **double** b = 26.32;
+	Adaugă două variabile	a + b = 36	36.439999	36.439999999999998
-	Scade variabila a doua din prima	b - a = 16	16.200001	16.200000000000003
*	Înmulțește două variabile	a * b = 260	266.358398	266.358399999999961
/	Împarte două variabile	b / a = 2	2.600791	2.600790513833992
%	Restul împărțirii pentru numere întregi	b % a = 6	Nu e cazul	Nu e cazul
++	Incrementează operatorul cu 1	a++ = 11	11.120000	11.119999999999999
--	Decrementează operatorul cu 1	a-- = 9	9.120000	9.119999999999999

Numere întregi

op_ar_int.c

```c
#include <stdio.h>

int main()
{
        int a = 10;
        int b = 26;

        printf ("\n--------Numere întregi---------------\n");
        printf ("a este: %d\n", a);
        printf ("b este: %d\n", b);
        printf ("----------------------\n\n");
        printf ("a + b este: %i\n", a + b);
        printf ("a - b este: %i\n", a - b);
        printf ("a * b este: %i\n", a * b);
        printf ("b / a este: %i\n", b / a);
        printf ("b %% a este: %i\n", b % a);
        a++;
        printf ("a++ este: %i\n", a);
        a = 10; /*readucem a la valoarea initiala */
        A--;
        printf ("a-- este: %i\n", a);
        return 0;
}
```

https://onlinegdb.com/Sk-51_4a7

```
--------Numere întregi---------------
a este: 10
b este: 26
----------------------

a + b este: 36
a - b este: -16
a * b este: 260
b / a este: 2
b % a este: 6
a++ este: 11
a-- este: 9
```

Numere flotante

op_ar_float.c

```c
#include <stdio.h>

int main()
{
        float a = 10.12;
        float b = 26.32;

        printf ("\n--------Numere flotante-------------\n");
        printf ("a este: %f\n", a);
        printf ("b este: %f\n", b);
        printf ("----------------------\n\n");

        printf ("a + b este: %f\n", a + b);
        printf ("b - a este: %f\n", b - a);
        printf ("a * b este: %f\n", a * b);
        printf ("b / a este: %f\n", b / a);
        a++;
        printf ("a++ este: %f\n", a);
        a=10.12; /* readucem a la valoarea initiala */
        A--;
        printf ("a-- este: %f\n", a);
        return 0;
}
```

https://onlinegdb.com/B18gluE6Q

```
--------Numere flotante-------------
a este: 10.120000
b este: 26.320000
----------------------

a + b este: 36.439999
b - a este: 16.200001
a * b este: 266.358398
b / a este: 2.600791
a++ este: 11.120000
a-- este: 9.120000
```

Operatori logici

Adevărat sau fals în C.

- Orice valoare care este diferită de 0 este considerată adevărată
- Valoarea 0 este considerată falsă
- Rezultatul operatorilor logici poate fi 0 (fals) sau 1 (adevărat).

Operator	Descriere	`int a = 1;` `int b = 0;`
&&	Operatorul AND. Daca ambele variabile sunt non-zero atunci rezultatul devine “adevărat” (1)	a && b este: 0
||	Operatorul OR. Dacă oricare dintre cele doua variabile este diferită de zero atunci rezultatul devine “adevărat” (1)	a || b este: 1
!	Operatorul NOT. Inversează condiția pe care o precede.	! a || b este: 0

Regulile se păstrează și pentru alt fel de tipuri de date - float, double.

Operatori logici (exemplu)

op_log.c

```c
#include <stdio.h>

int main()
{
     int a = 1;
     int b = 0;
     int c = 12;
     int d = 0;

     printf ("\n---------------------\n");
     printf ("a este: %d\n", a);
     printf ("b este: %d\n", b);
     printf ("c este: %d\n", c);
     printf ("d este: %d\n", d);
     printf ("\n---------&&--------------\n");
     printf ("a && b este: %i\n", (a && b));
     printf ("a && c este: %i\n", (a && c));
     printf ("b && d este: %i\n", (b && d));
     printf ("\n----------||-------------\n");
     printf ("a || b este: %i\n", (a || b));
     printf ("a || c este: %i\n", (a || c));
     printf ("b || d este: %i\n", (b || d));
     printf ("\n------------! &&----------\n");
     printf ("! a || b este: %i\n", !(a || b));
     printf ("! a || c este: %i\n", !(a || c));
     printf ("! b || d este: %i\n", !(b || d));
     return 0;
}
```

https://onlinegdb.com/H1jLedVTm

```
---------------------
a este: 1
b este: 0
c este: 12
d este: 0

---------&&--------------
a && b este: 0
a && c este: 1
b && d este: 0

----------||-------------
a || b este: 1
a || c este: 1
b || d este: 0

------------! &&----------
! a || b este: 0
! a || c este: 0
! b || d este: 1
```

Operatori relaționali

Rezultatul operatorilor relaționali poate fi 0 (fals) sau 1 (adevărat).

Operator	Descriere	**int** a = 10; **int** b = 26;
==	Verifică dacă cele două valori sunt egale sau nu. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)	a == b este: 0
!=	Verifică dacă cele două valori sunt diferite. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)	a != b este: 1
>	Verifică dacă valoarea variabilei din stânga este mai mare decât cea a celei din dreapta. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)	a > b este: 0
<	Verifică dacă valoarea variabilei din stânga este mai mică decât cea a celei din dreapta. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)	a < b este: 1
>=	Verifică dacă valoarea variabilei din stânga este mai mare SAU EGALĂ decât cea a celei din dreapta. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)	a >= b este: 0
<=	Verifică dacă valoarea variabilei din stânga este mai mică SAU EGALA decât cea a celei din dreapta. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)	a <= b este: 1

Exemplu simplu

op_rel_int.c

```c
#include <stdio.h>

int main()
{
   int a = 10;
   int b = 26;

   printf ("a == b este: %i\n", (a == b));
   printf ("a != b este: %i\n", (a != b));
   printf ("a > b este: %i\n", (a > b));
   printf ("a < b este: %i\n", (a < b));
   printf ("a >= b este: %i\n", (a >= b));
   printf ("a <= b este: %i\n", (a <= b));
   return 0;
}
```

https://onlinegdb.com/rJRje_N6m

```
a == b este: 0
a != b este: 1
a > b este: 0
a < b este: 1
a >= b este: 0
a <= b este: 1
```

Float-ul lovește din nou

op_rel_float.c

```c
#include <stdio.h>
#include <limits.h>

int main()
{
  float a = 1.1;
  float b = 2.2;
  float c = 3.3;
  float d = a + b;

  printf ("a este %f\n", a);
  printf ("b este %f\n", b);
  printf ("c este %f\n", c);
  printf ("d este %f\n", d);
  printf ("c == d este: %i\n", (c == d));
  printf ("\n--------------------------\n");
  printf ("dimensiunea lui a este: %zu octeti\n", sizeof(a));
  printf ("dimensiunea lui b este: %zu octeti\n", sizeof(b));
  printf ("dimensiunea lui c este: %zu octeti\n", sizeof(c));
  printf ("dimensiunea lui d este: %zu octeti\n", sizeof(d));
  return 0;
}
```

https://onlinegdb.com/SJNbZdNTQ

```
a este 1.100000
b este 2.200000
c este 3.300000
d este 3.300000
c == d este: 0

---------------------------
dimensiunea lui a este: 4 octeti
dimensiunea lui b este: 4 octeti
dimensiunea lui c este: 4 octeti
dimensiunea lui d este: 4 octeti
```

Atunci când 1.1 nu e egal cu 1.1 ...

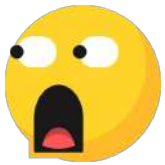
op_rel_float_2.c

```c
#include <stdio.h>
#include <limits.h>

int main()
{
   float a = 1.1;

   printf ("a este %f\n", a);
   printf ("dimensiunea lui a este: %zu\n", sizeof(a));

   printf ("\n---------------------------\n");
   printf ("a == 1.1 este: %i\n", (a == 1.1));
   printf ("a == 1.1f este: %i\n", (a == 1.1f));
   printf ("dimensiunea lui 1.1 este: %zu\n", sizeof(1.1));
   printf ("dimensiunea lui 1.1f este: %zu\n", sizeof(1.1f));
   return 0;
}
```

https://onlinegdb.com/ByoOWOEaX

```
a este 1.100000
dimensiunea lui a este: 4

---------------------------
a == 1.1 este: 0
a == 1.1f este: 1
dimensiunea lui 1.1 este: 8
dimensiunea lui 1.1f este: 4
```

float(1.1)	1.100000023841857910156 25
double(1.1)	1.100000000000000088817841970012523233890533447265625



Sau când 0,7 nu e egal cu 0.7 ...

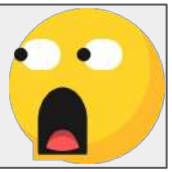
	semn		exponent		mantisa		total
float	0	1b	01111110	8b	01100110011001100110011	23b	32 biți
double	0	1b	01111111110	11b	0110011001100110011001100110011001100110011001100110	52b	64 biți

float	0.69999998807907104492187 5
double	0.69999999999999999555910790149 9

Valorile stocate în realitate

```
#include <stdio.h>

void main()
{
    float a = 0.7;
    if (a == 0.7) {
        printf("egale");
    } else {

printf("inegale");
    }
}
```

inegale



definim variabila a de tip float și încercăm să stocăm 0.7 în ea. În realitate stocăm 0.69999998807907104492187 5

0.7 în condiția if devine o **variabilă anonimă**. Ce tip are? float, double? Programatorul nu a asociat un tip dar 0.7 din if trebuie să fie stocat în memorie și pentru aceasta compilatorul **trebuie să decidă un tip** doar pe baza evaluării numărului 0.7. Compilatorul vede că nu este număr întreg și alege **cel mai mare tip flotant comun** (double).

deci a == 0.7 compară valoarea lui 0.7 float (stocată în variabila a) cu valoarea lui 0.7 double, stocată în variabila anonimă.

a **este automat promovat la double** (dar nu 0.7 ci 0.69999998807907104492187 5) și comparat cu 0.7 stocat în double (0.69999999999999999555910790149 9). Cele două valori NU sunt egale.

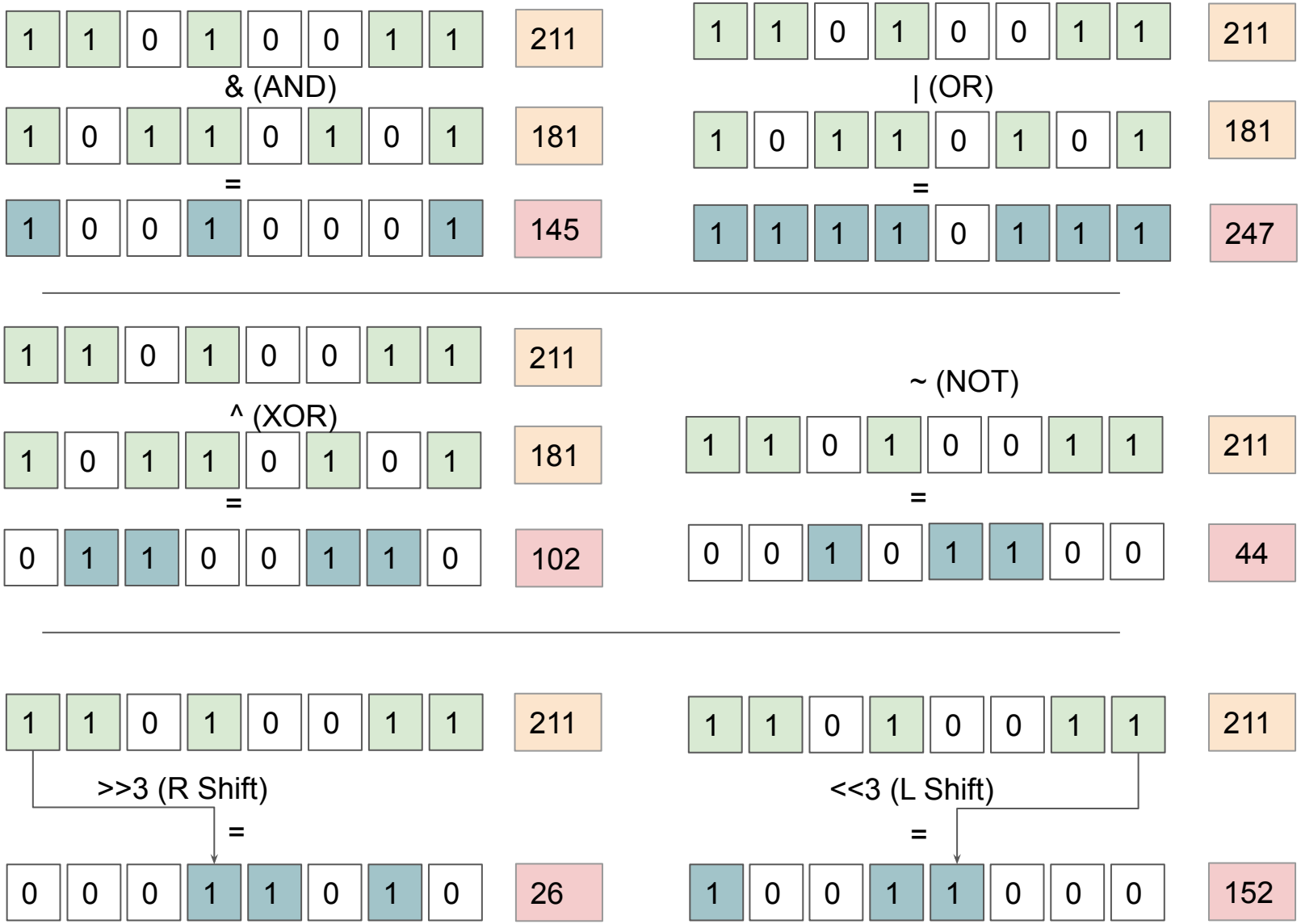
Dacă scriam `a == 0.7f` (ca în slide-ul anterior) atunci îi comunicam compilatorului că 0.7 este float și atunci nu ar fi fost stocat automat ca double. Egalitatea ar fi fost adevărată.

Operatori la nivel de bit

Operatorii la nivel de bit operează direct cu fiecare bit al variabilelor.

Operator	Descriere
&	AND binar - copiază un bit în rezultat dacă există în ambele variabile
|	OR binar - copiază un bit dacă există în oricare dintre variabile
^	Exclusive OR (XOR)
~	NOT (Inversează fiecare bit)
<<	Left Shift - mută patternul de biți cu n caractere la stânga
>>	Right Shift - mută patternul de biți cu n caractere la dreapta

Operatori la nivel de bit



Operatori la nivel de bit

op_bitwise.c

```c
#include <stdio.h>
#include <limits.h>

int main()
{
   unsigned char a = 211;
   unsigned char b = 181;
   unsigned char c =  ~a;
   unsigned char d =  a<<3;

   printf ("a este: %i\n", a );
   printf ("b este: %i\n", b);
   printf ("-------------------------\n");

   printf ("Binary AND este: %i\n", a & b);
   printf ("Binary OR este: %i\n", a | b);
   printf ("Binary XOR este: %i\n", a ^ b);
   printf ("Binary Ones pentru a este: %i\n", c);
   printf ("Left shift 3 pentru a este: %i\n", d);
   printf ("Right shift 3 pentru a este: %i\n",
a>>3);

   return 0;
}
```

```
a este: 211
b este: 181
-------------------------
Binary AND este: 145
Binary OR este: 247
Binary XOR este: 102
Binary Ones pentru a este: 44
Left shift 3 pentru a este: 152
Right shift 3 pentru a este: 26
```

https://onlinegdb.com/By0ZzOETQ

Verificarea parității unui număr cu AND

1	0	1	0	0	1	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	1	1

128 + 32 + 4 + 1 = 165 Număr impar

Orice număr impar va avea ultimul bit (2^0) setat pe 1, indiferent de valorile celorlalți biti

1	0	1	0	0	1	0	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	1	1

128 + 32 + 4 = 164 Număr par

1	0	1	0	0	1	0	1	165
& (AND)								&
0	0	0	0	0	0	0	1	1
=								=
0	0	0	0	0	0	0	1	1

1	0	1	0	0	1	0	0	164
& (AND)								&
0	0	0	0	0	0	0	1	1
=								=
0	0	0	0	0	0	0	0	0

Inversarea a două valori cu XOR

xor_switch.c

xor_switch_print.c

```c
#include <stdio.h>

int main()
{
   char a = 13;
   char b = 28;
   printf ("inainte: a: %d, b: %d\n", a,b);

   a = a ^ b;
   b = b ^ a;
   a = a ^ b;

   printf ("dupa: a: %d, b: %d\n", a,b);

}
```

https://onlinegdb.com/HJ9TatFYv

```
inainte: a: 13, b: 28
dupa: a: 28, b: 13
```

a	0	0	0	0	1	1	0	1
b	0	0	0	1	1	1	0	0
a=a^b	0	0	0	1	0	0	0	1
b	0	0	0	1	1	1	0	0
a	0	0	0	1	0	0	0	1
b = b^a	0	0	0	0	1	1	0	1
a = a^b	0	0	0	1	1	1	0	0
b	0	0	0	0	1	1	0	1

Codul cu print binar:
https://onlinegdb.com/BJywB7pFP

Numere binare folosind bit shift

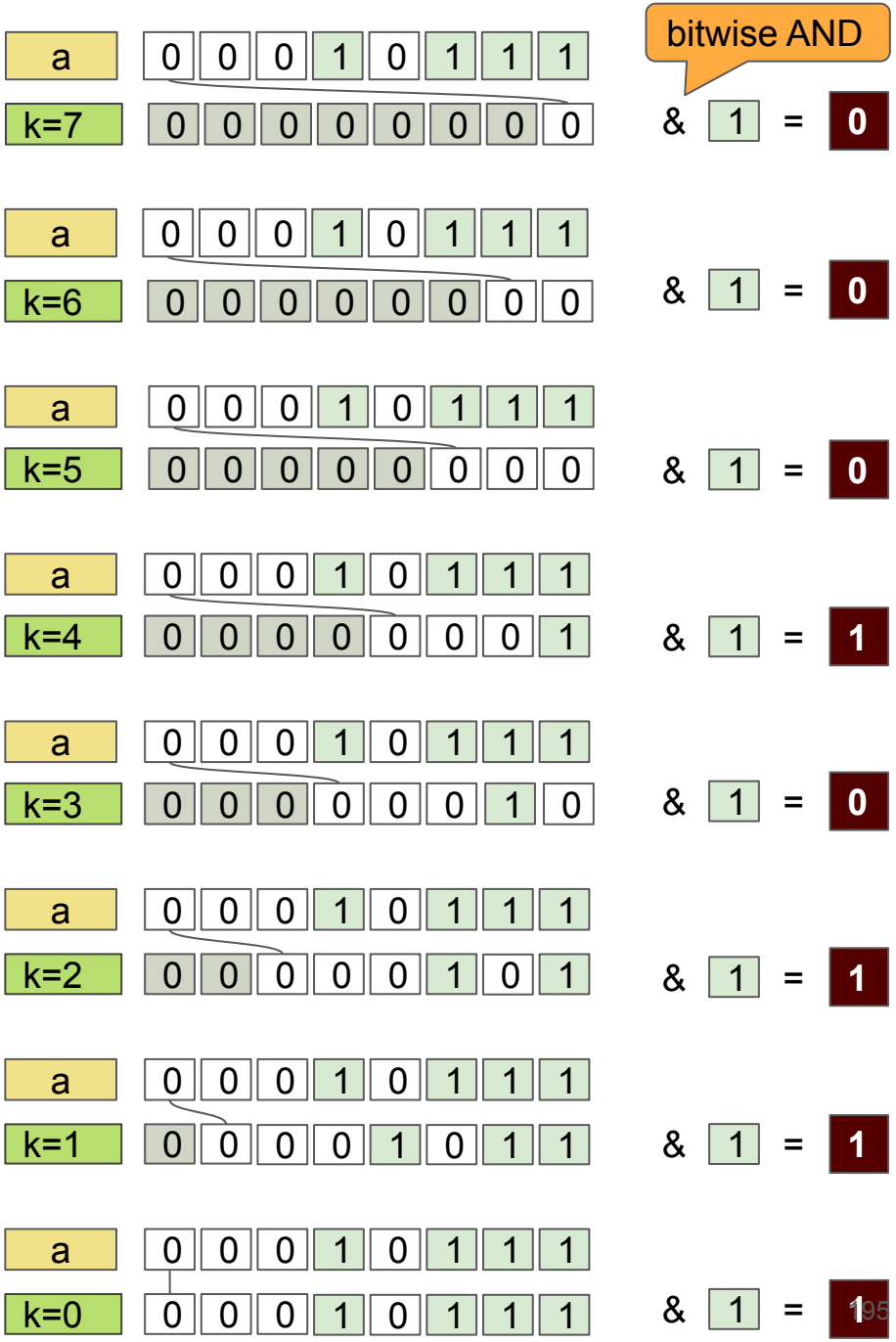
bit_shift_bin.c

```c
#include <stdio.h>

void printBinChar(char c);

int main()
{
    char a = 23;
    printBinChar(a);
    return 0;
}

void printBinChar(char n)
{
    int k;
    int i = 8;
    for (k = i - 1; k >= 0; k--) {
        if ((n >> k) & 1)
            printf("1");
        else
            printf("0");
    }
    printf("\n");
}
```

https://onlinegdb.com/SJWqyqFFw

Pentru urmărirea bit-shiftului folositi urmatorul cod: https://onlinegdb.com/SJe9uPcYFD

bit_shift_bin_expl.c



Operatori de atribuire

Op	Descriere	Exemplu
=	Atribuie valoarea din partea dreapta a operatorului variabilei din partea stângă	a = b + c
+=	Adaugă valoarea variabilei din dreapta celei din stânga și atribuie valoarea variabilei din stânga.	c += a este la fel cu c = c + a
-=	Scade valoarea variabilei din partea dreaptă din cea din stânga și atribuie rezultatul variabilei din partea stângă.	c -= a este la fel cu c = c - a
*=	Înmulțește valoarea variabilei din stânga operatorului cu valoarea celei din stânga și atribuie rezultatul variabilei din stânga.	c *= a este la fel cu c = c * a
/=	Împarte valoarea variabilei din stânga la cea din dreapta și atribuie rezultatul variabilei din stânga.	c /= a este la fel cu c = c / a
%=	Atribuie variabilei din stânga valoarea modulului împărțirii valorii din stânga la cea din dreapta	c %= a este la fel cu c = c % a
<<=	Left shift si atribuire	c <<= 2 este la fel cu c = c<<2
>>=	Right shift și atribuire	c <<= 2 este la fel cu c = c<<2
&=	Bitwise AND și atribuire	c &= a este la fel cu c = c & a
^=	Bitwise XOR și atribuire	c ^= a este la fel cu c = c ^ a
|=	Bitwise OR și atribuire	c |= a este la fel cu c = c | a

Operatori de atribuire - exemple

op_atribuire.c

```c
#include <stdio.h>

int main()
{
   int a = 211;
   int b = 100;
   printf ("a este: %i\n", a );
   printf ("b este: %i\n", b);
   printf ("-------------------------\n");
   printf ("b += a este: %i\n", b += a);
   printf ("b este acum: %i\n", b);
   b = 100;
   printf ("b -= a este: %i\n", b -= a);
   printf ("b este acum: %i\n", b);
   b = 100;
   printf ("b *= a este: %i\n", b *= a);
   printf ("b este acum: %i\n", b);
   b = 100;
   printf ("a /= b (intreg) este: %i\n", a /= b);
   printf ("a este acum: %i\n", a);
   a = 211;
   printf ("a %%= b este: %i\n", a %= b);
   printf ("a este acum: %i\n", a);
   return 0;
}
```

```
a este: 211
b este: 100
-------------------------
b += a este: 311
b este acum: 311
b -= a este: -111
b este acum: -111
b *= a este: 21100
b este acum: 21100
a /= b (intreg) este: 2
a este acum: 2
a %= b este: 11
a este acum: 11
```

https://onlinegdb.com/SkMKzuNpX

Alți operatori

Operator	Descriere	Exemple
sizeof()	Returnează dimensiunea unei variabile.	sizeof(a), un număr
&	Returnează adresa unei variabile	&a = 0x7fffa9a5a92c
*	Returnează un pointer către o variabilă	*a;
?:	Expresie condițională	a > 23 ? 1 : 0 returnează 1 dacă variabila a conține un număr mai mare decât 23 și 0 dacă nu.

Alți operatori - exemplu (sizeof, &, ternary)

op_misc.c

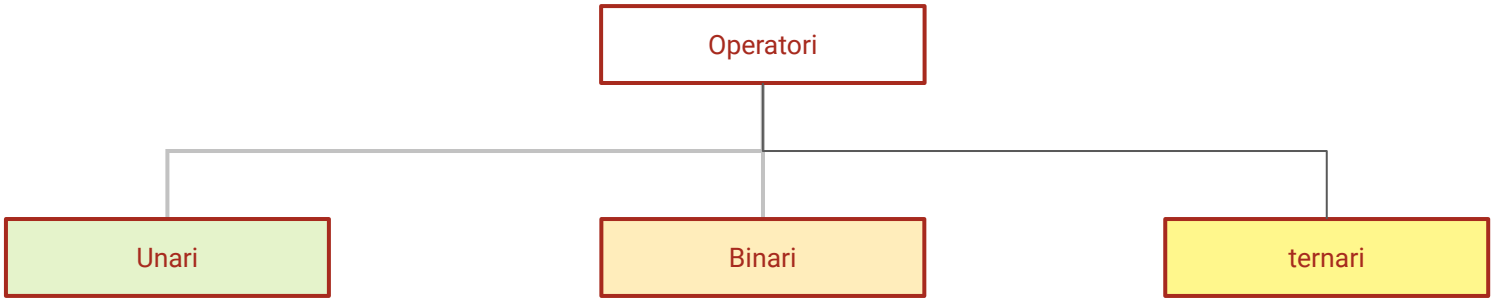
```c
#include <stdio.h>

int main()
{
   int a = 211;
   printf ("a este: %i\n", a );
   printf ("--------------------------\n");
   printf ("Dimensiunea variabilei a este: %zu \n", sizeof(a));
   printf ("Adresa de memorie a variabilei a este: %p \n", &a);
   printf ("a este mai mare decat 23: %i \n", (a > 23?1:0));
   return 0;
}
```

https://onlinegdb.com/SkyN7_VTQ

```
a este: 211
--------------------------
Dimensiunea variabilei a este: 4
Adresa de memorie a variabilei a este: 0x7fffa9a5a92c
a este mai mare decat 23: 1
```

Operatori unari, binari, ternari

- Operatorii unari implică o singură variabilă.
- Operatorii binari implică două variabile.
- Operatorii ternari implică trei variabile.



- sizeof
- incrementare
  - post (a = x++)
  - pre (a = ++X)
- decrementare
  - pre (a = x--)
  - post (a = --x)

- aritmetic (+,-,*,/,%)
- relațional (==,!=,>,<,>=,<=)
- logic (&&,||,!)
- bitwise (&,^,|,~,<<,>>)
- atribuire (+,+=,-=...)
- speciali (&,*)

condițional: (?:)

Operatori unari - exemplu

op_unary.c

```c
#include <stdio.h>

int main()
{
   int a, b;
   a = 10;
   b = a++;
   printf("Postincrement\n");
   printf ("a este: %i\n",a);
   printf ("b este: %i\n",b);
   a = 10;
   b = ++a;
   printf("Preincrement\n");
   printf ("a este: %i\n",a);
   printf ("b este: %i\n",b);
   a = 10;
   b = a--;
   printf("Postdecrement\n");
   printf ("a este: %i\n",a);
   printf ("b este: %i\n",b);
   a = 10;
   b = --a;
   printf("Predecrement\n");
   printf ("a este: %i\n",a);
   printf ("b este: %i\n",b);
   return 0;
}
```

https://onlinegdb.com/BJ-YQON6Q

```
Postincrement
a este: 11
b este: 10
Preincrement
a este: 11
b este: 11
Postdecrement
a este: 9
b este: 10
Predecrement
a este: 9
b este: 9
```

Postincrement și postdecrement se aplică după atribuire

Preincrement și predecrement se aplică înainte de atribuire

Precedența operatorilor

Operator	Asociere
!,++,--, - unar	dreapta la stânga
*,/,%	stânga la dreapta
<, <=, > >=	stânga la dreapta
==, !=	stânga la dreapta
&&	stânga la dreapta
||	stânga la dreapta

Precedența operatorilor - exemplu

op_precedenta.c

```c
#include <stdio.h>

int main() {

  int a = 20;
  int b = 10;
  int c = 15;
  int d = 5;
  int e;

  e = a + b * c / d;        // 20 +  15
  printf("Valoarea a + b * c / d este : %d\n",  e );

  e = (a + b) * c / d;       // ( 30 * 15 ) / 5
  printf("Valoarea (a + b) * c / d este : %d\n",  e );

  e = ((a + b) * c) / d;     // (30 * 15 ) / 5
  printf("Valoarea ((a + b) * c) / d este  : %d\n" ,  e );

  e = (a + b) * (c / d);    // (30) * (15/5)
  printf("Valoarea (a + b) * (c / d) este  : %d\n",  e );

  e = a + (b * c) / d;      //  20 + (150/5)
  printf("Valoarea a + (b * c) / d este  : %d\n" ,  e );

  return 0;
}
```

```
Valoarea a + b * c / d este : 50
Valoarea (a + b) * c / d este : 90
Valoarea ((a + b) * c) / d este  : 90
Valoarea (a + b) * (c / d) este  : 90
Valoarea a + (b * c) / d este  : 50
```

https://onlinegdb.com/Skma7ONpm

Instrucțiuni decizionale

Blocuri de cod

C (ca multe alte limbaje) permite definirea unor blocuri de cod care limitează vizibilitatea (eng. scope) variabilelor din interiorul acestora (cele declarate cu clasa de stocare auto). Acest comportament se numeste “lexical scoping”.

```c
#include <stdio.h>


int main() {
  int a = 10;
  {
    int b = 18;
    printf("b in bloc este %i\n", b);
    {
      int c = 23;
      printf("b,c in bloc:  %i,%i\n", b,c);
    }
  }
   int d = 20;
   printf("a este %i\n", a);
   printf("b este %i\n", b);
   printf("d este %i\n", d);
   return 0;
}
```

bloc 1

bloc 2

codeblocks.c

```c
#include <stdio.h>


int main() {
   int a = 10;
   {
     int b = 18;
     printf("b in bloc este %i\n", b);
     {
       int c = 23;
       printf("b,c in bloc:  %i,%i\n", b,c);
     }
   }
   int d = 20;
   printf("a este %i\n", a);
//    printf("b este %i\n", b);
   printf("d este %i\n", d);
   return 0;
}
```

https://onlinegdb.com/SJ2b2KKT7

```
main.c: In function 'main':
main.c:16:26: error: 'b' undeclared (first use
in this function)
    printf("b este %i\n", b);
```

```
b in bloc este 18
b,c in bloc:  18,23
a este 10
d este 20
```

Cum să luăm decizii

Deciziile sunt instrucțiuni care execută anumite blocuri de cod în funcție de răspunsul la o anumită întrebare. Întrebarea este evaluată și poate avea două răspunsuri: adevărat sau fals.

Adevărat sau fals in C.

- Orice valoare care este diferită de 0 este considerată adevărată
- Valoarea 0 este considerată falsă
- Rezultatul operatorilor logici poate fi 0 (fals) sau 1 (adevărat).

```c
int main()
{
   int N;
   printf("introduceti un numar\n");
   scanf("%i",&N);

   if (N > 0) {
       printf("Numarul e mai mare decat zero\n");
   } else {
       printf("Numarul e mai mic decat zero\n");
   }
   return 0;
}
```

```
introduceti un numar
12
Numarul e mai mare decat zero
```

```
introduceti un numar
-432
Numarul e mai mic decat zero
```

https://onlinegdb.com/B1RbH623X

if-else

Cu paranteze

```c
int main()
{
    int N;
    printf("numar ?\n");
    scanf("%i",&N);

    if (N > 0) {
        printf("> decat zero\n");
    } else {
        printf("< decat zero\n");
    }

    return 0;
}
```

https://onlinegdb.com/Sy2fbSw6X

Fără paranteze

if-no-braces.c

```c
int main()
{
  int N;
  printf("numar? \n");
  scanf("%i",&N);

  if (N > 0)
      printf("> decat zero\n");
  else
      printf("< decat zero\n");

   printf("restul ...");
  return 0;
}
```

https://onlinegdb.com/HyH9brDpm

În cazul în care după if sau else urmează o singură instrucțiune (doar una), se pot omite parantezele. Dar aceea trebuie sa fie instrucțiune și nu declarație.

Editoarele special concepute pentru scrierea codului vor pune paranteze și în cazul în care există o singură instrucțiune. Chiar dacă nu sunt necesare, acestea fac codul mult mai ușor de înțeles și urmărit.

Apple goto fail SSL bug

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
   goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
   goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
   goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
   goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
   goto fail;
   goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
   goto fail;

err = sslRawVerify(...);
. . .
```

About the security content of iOS 7.0.6

This document describes the security content of iOS 7.0.6.

iOS 7.0.6

- Data Security

Available for: iPhone 4 and later, iPod touch (5th generation), iPad 2 and later

Impact: An attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS

Description: Secure Transport failed to validate the authenticity of the connection. This issue was addressed by restoring missing validation steps.

CVE-ID

CVE-2014-1266

Codul va sări peste funcția **sslRawVerify** în toate situațiile, permițând conexiunea SSL **fără verificarea reală a certificatelor**.

În Februarie 2014, Apple a publicat un update de securitate pentru toate dispozitivele mobile, în categoria "Data Security":
Description: Secure Transport failed to validate the authenticity of the connection. This issue was addressed by restoring missing validation steps.

Lipsa parantezelor poate fii "cool" dar poate avea consecințe dramatice. Există standarde de cod care încurajează evitarea parantezelor acolo unde este posibil, există altele care obligă la prezența parantezelor în toate cazurile.

https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/

if-else

```c
int main()
{
  int N;
  printf("numar? \n");
  scanf("%i",&N);

  if (N > 0)
      printf("> decat zero\n");
  else
      printf("< decat zero\n");

   printf("restul ...");
  return 0;
}
```

```c
int main()
{
  int N;
  printf("numar? \n");
  scanf("%i",&N);

  if (N > 0)
      int a = 2;
  else
      printf("< decat zero\n");

  printf("restul ...");
  return 0;
}
```

```c
int main()
{
  int N;
  printf("numar? \n");
  scanf("%i",&N);

  if (N > 0)
     {
      int a = 2;
     }
  else
      printf("< decat zero\n");

  printf("restul ...");
  return 0;
}
```

main.c: In function ‘main’:
main.c:10:7: error: expected expression before ‘int’
    int a = 2;

Declarațiile nu sunt permise după if

Blocul de cod este permis după if,
in el se pot face declaratii

Alcătuirea condiției if

Operator	Descriere
==	Verifică dacă cele două valori sunt egale sau nu. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)
!=	Verifică dacă cele două valori sunt diferite. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)
>	Verifică dacă valoarea variabilei din stânga este mai mare decât cea a celei din dreapta. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)
<	Verifică dacă valoarea variabilei din stânga este mai mică decât cea a celei din dreapta. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)
>=	Verifică dacă valoarea variabilei din stânga este mai mare SAU EGALĂ decât cea a celei din dreapta. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)
<=	Verifică dacă valoarea variabilei din stânga este mai mică SAU EGALA decât cea a celei din dreapta. Dacă da, condiția devine 1 (adevărată). Dacă nu, condiția devine 0 (falsă)

```
if (a == 3)
```

```
if (a > 3)
```

```
if (a >= 3)
```

```
if (a != 3)
```

```
if (a < 3)
```

```
if (a <= 3)
```

if-else - condiții complexe

Operator	Descriere
&&	Operatorul AND. Daca ambele variabile sunt non-zero atunci rezultatul devine “adevarat” (1)
||	Operatorul OR. Daca oricare dintre cele doua variabile este diferita de zero atunci rezultatul devine “adevarat” (1)
!	Operatorul NOT. Inverseaza condiția pe care o precede.

```
if (a > 3 && a < 8 )
```

```
if ((a > 3 && a < 8 ) || (b > 19))
```

```
if (a > 3 || a < 8 )
```

```
if (!(a > 3 && a < 8 ) || (b > 19))
```

```
if (a == 3 || a > 8 )
```

```
if (!(a > 3 || a < 8 ))
```

if-else imbricate

Se pot imbrica oricâte condiții if una în cealaltă. De la un punct încolo, codul devine greu de urmărit și trebuie utilizate alte moduri de programare (ex. funcții).

```c
#include <stdio.h>

int main()
{
    int a = 4;

    if (a > 3 && a < 8) {
        printf("check\n");
    }

    if (a > 3) {
        if (a < 8) {
            printf("check\n");
        }
    }
    return 0;
}
```

if-imbricat.c

https://onlinegdb.com/H1-0-SPpX

```
check
check
```

Se pot folosi instrucțiuni suplimentare în interiorul unui if, și în afara if-urilor imbricate.

```c
#include <stdio.h>

int main()
{
    int a = 4;
    if (a > 3) {
        printf("Inainte de if\n");
        if (a < 8) {
            printf("check\n");
        }
        printf("Dupa if\n");
    }
    return 0;
}
```

if-imbricat_2.c

https://onlinegdb.com/HkcvMSP6m

```
Inainte de if
check
Dupa if
```

else if

if-else_if.c

```c
int main()
{
    int N;
    printf("introduceti un numar\n");
    scanf("%i",&N);

    if (N >= 10) {
        printf("Numarul este mai mare sau egal cu 10\n");
    } else if (N < 10 && N >= 0) {
        printf("Numarul este mai mic decat 10\n");
    } else {
        printf("Numarul este negativ\n");
    }
    return 0;
}
```

https://onlinegdb.com/SySpfrPp7

```
introduceti un numar
32
Numarul este mai mare sau egal cu 10
```

```
introduceti un numar
7
Numarul este mai mic decat 10
```

```
introduceti un numar
-21
Numarul este negativ
```

Variabile locale

```c
#include <stdio.h>

int main()
{
  int N = 3;
  int a = 2;
  if (N >= 10) {
      a = 100;
      printf("N >= cu 10\n");
  } else if (N < 10 && N >= 0) {
      printf("N <= 10\n");
  } else {
      printf("N negativ\n");
  }
   printf("a dupa if %d\n", a);

  return 0;
}
```

if-imbricat_2.c

https://onlinegdb.com/By4TQSDT7

N <= 10
a dupa if 2

```c
#include <stdio.h>

int main()
{
  int N = 13;
  int a = 2;
  if (N >= 10) {
      a = 100;
      printf("N >= cu 10\n");
  } else if (N < 10 && N >= 0) {
      printf("N <= 10\n");
  } else {
      printf("N negativ\n");
  }
   printf("a dupa if %d\n", a);

  return 0;
}
```

Modificăm N

if-imbricat_2.c

https://onlinegdb.com/rJB4VSv6Q

N >= cu 10
a dupa if 100

Dacă programul ajunge la ramura care atribuie valoarea 100 variabilei a (definită în afara if-ului), variabila își menține valoarea după if.

Variabile locale

```c
#include <stdio.h>

int main()
{
  int N = 13;
  int a = 2;
  if (N >= 10) {
      a = 100;
      int b = 12;
      printf("N >= cu 10\n");
  } else if (N < 10 && N >= 0) {
      printf("N <= 10\n");
  } else {
      printf("N negativ\n");
  }
   printf("a dupa if %d\n", a);
   printf("b dupa if %d\n", b);
  return 0;
}
```

if-imbricat_3.c

https://onlinegdb.com/rJqIrBDam

```
main.c:17:30: error: 'b' undeclared (first use in this function)
    printf("b dupa if %d\n", b);
```

Variabila b nu este definită în afara if-ului

```c
#include <stdio.h>

int main()
{
  int N = 13;
  int a = 2;
  if (N >= 10) {
      a = 100;
      int b = 12;
      printf("N >= cu 10\n");
  } else if (N < 10 && N >= 0) {
      printf("N <= 10\n");
  } else {
      printf("N negativ\n");
  }
   printf("a dupa if %d\n", a);

  return 0;
}
```

if-imbricat_4.c

https://onlinegdb.com/S1e4PBDa7

```
N >= cu 10
a dupa if 100
```

Dacă b nu este apelată în afara blocului în care a fost definită, programul se compilează

ternary-hell.c

```c
#include<stdio.h>

int main()
{
  int a = 23;
   int b = 44;
   int c = 99;
   int max1 = a>b ? a>c?a:c : b>c?b:c;
   int max = 0;
   if (a > b) {
       if (a > c) {
           max = a;
       } else {
           max = c;
       }
   } else {
       if (b > c) {
           max = b;
       } else {
           max = c;
       }
   }
   printf ("Max 1: %i\n", max1);
   printf ("Max 2: %i\n", max);
   return 0;
}
```

https://onlinegdb.com/H13zUtsTX

```c
a>b ? a>c?a:c : b>c?b:c;
```

```
Max 1: 99
Max 2: 99
```

switch - case

switch-case.c

```c
int main () {
  char gr = 'B';
  switch(gr) {
     case 'A' :
        printf("%c: Excelent!\n", gr );
        break;
     case 'B' :
        printf("%c: Close, but no cigar\n", gr );
        break;
     case 'C' :
        printf("%c: On the top of gauss hat\n", gr);
        break;
     case 'D' :
        printf("%c: Still in school, not for long\n", gr);
        break;
     case 'F' :
        printf("%c: you need a job\n", gr);
        break;
     default :
        printf("%c: Invalid grade\n", gr);
  }
  return 0;
}
```

https://onlinegdb.com/HyBADBDpQ

```
B: Close, but no cigar
```

switch - case

```
switch(expresie) {

   case constanta  :
      instructiuni;
      break; /* optional */

   case constanta  :
      instructiuni;
      break; /* optional */

   default : /* Optional */
   instructiuni(s);
}
```

expresie trebuie să poată lua valoarea unui număr întreg sau caracter prin conversie către ASCII

constanta care specifica fiecare caz trebuie sa fie **o singură valoare** (nu se pot folosi construcții de genul “case (a < 5)”

tipul de data folosită în **constantă** trebuie să fie **același** cu cel din **expresie**

cand se întâlnește un caz adevărat, se vor executa toate instrucțiunile întâlnite **până la următorul break**.

break-ul e opțional din punct de vedere al sintaxei dar este obligatoriu dacă se dorește separarea codului executat la fiecare caz.

switch-ul poate avea un caz numit **default** care se va executa dacă nici unul dintre celelalte cazuri nu este adevărat sau dacă acel caz care este adevărat nu are instrucțiunea break.

switch - case

switch-case_2.c

```c
int main () {
  char gr = 'B';
  switch(gr) {
     case 'A' :
     case 'B' :
     case 'C' :
        printf("%c: OK\n", gr);
        break;
     case 'D' :
     case 'F' :
        printf("%c: Not OK\n", gr);
        break;
     default :
        printf("%c: Invalid grade\n", gr);
  }
  return 0;
}
```

https://onlinegdb.com/r1VndrPa7

Se pot grupa mai multe cazuri pentru același cod.

De aceea switch-ul trece la următorul caz în lipsa break-ului.

```
A: OK
```

```
B: OK
```

```
C: OK
```

```
D: Not OK
```

```
F: Not OK
```

switch - case în alte limbaje de programare

switch-case.php

```php
<?php
$i = 5;

switch ($i) {
   case 0:
       echo "i este egal cu 0";
       break;
   case 1:
       echo "i este egal cu 1";
       break;
   case 2:
       echo "i este >= cu 2";
       break;
   case ($i >= 4 || $i <= 6):
       echo "i este intre 4 si 6";
       break;

   default:
       echo "i este in alta parte";
}
?>
```

https://onlinegdb.com/BklNtrPTm

```
i este intre 4 si 6
```

În php cazurile pot conține expresii logice.

Dar, dacă se folosește break la toate nivelele, acestea trebuie sa fie exclusive (nici o valoare a lui i nu trebuie sa poată intra în mai mult de un caz).

Exista și alte limbaje de programare care permit expresii în construcțiile de tip switch/case (mai toate versiunile de SQL, ruby, basic.

Compilatorul gcc implementează extensii la instrucțiunea switch/case și permite folosirea intervalelor:

```
case 'A' ... 'Z':

case 1 ... 3:
```

switch - case de față cu if - else if

- Orice construcție de tip **switch/case** poate fi scrisă folosind **if/else if**.

- **Nu orice construcție** de tip if/else if poate fi scrisă cu switch/case (pentru că nu putem pune expresii la cazuri în c).

- Switch/case este în general **mai rapid** decât aceeași construcție if/else (deși depinde de compilator și de construcție în sine).

- Uneori switch/case generează cod mai mare decat if/else if()
  `/* A bunch of if...else if... gives smaller code than switch...case ! */` (ARDUINO)

- Construcțiile de tip switch/case sunt **mai clare și mai ușor de citit** decat if/else.

- În general switch/case sunt folosite la listele lungi, cu multe cazuri și cu condiții simple (ex: coduri de eroare diferite care declanșează comportamente diferite).

- If/else if se folosește mai degrabă în situațiile în care sunt cazuri puține cu condiții complexe.

Repetiții

Cum să repetăm aceleași instrucțiuni de mai multe ori

pseudocod

```
for cnt = 1, N, 1
    ...
end for
```

```
while cnt <= N
    …
end while
```

```
do
    ...
while count <= N
```

cod în limbajul C

```c
int main () {

  int a;

  for( a = 10; a < 15; a = a + 1 ){
     printf("a = %d\n", a);
  }

  return 0;
}
```

```c
int main () {

  int a = 10;

  while( a < 15 ) {
     printf("a = %d\n", a);
     a++;
  }
  return 0;
}
```

```c
int main () {

  int a = 10;

  do {
     printf("a = %d\n", a);
     a = a + 1;
  } while( a < 15 );
  return 0;
}
```

```
a = 10
a = 11
a = 12
a = 13
a = 14
```

```
a = 10
a = 11
a = 12
a = 13
a = 14
```

```
a = 10
a = 11
a = 12
a = 13
a = 14
```

Bucla for

sintaxa instrucțiunii for:

```
for ( init; condition; increment )
{
   statement(s);
}
```

Bucla for are incrementarea inclusă în declarație

```
for( a = 10; a < 15; a = a + 1 )
{
     printf("a = %d\n", a);
}
```

```
int main () {

  int a;

  for( a = 10; a < 15; a = a + 1 ){
     printf("a = %d\n", a);

  }
  return 0;
}
```

```
a = 10
a = 11
a = 12
a = 13
a = 14
```

```
int main () {

  int a;

  for( a = 10; a < 15; a = a + 1 ){
     printf("a = %d\n", a);
     a++;
  }
  return 0;
}
```

!

```
a = 10
a = 12
a = 14
```

Bucla for

Se pot omite parantezele și la bucla for (adică se poate renunța la declararea unui bloc de cod) dacă instrucțiunea care trebuie repetată este una singura

```
int main () {

 int a;

 for( a = 10; a < 15; a = a + 1 )
   printf("a = %d\n", a);
   printf("a afara = %d\n", a);

 return 0;
}
```

```
a = 10
a = 11
a = 12
a = 13
a = 14
a afara = 15
```

Observații:

Cea de-a doua instrucțiune printf se execută o singură dată, după terminarea buclei. În lipsa declarării unui bloc de cod, bucla are efect exclusiv asupra primei instrucțiuni de după for.

Valoarea variabilei **a** după for este 15. Aceasta e valoarea la care condiția (a < 15) devine adevărată și bucla se oprește.

Codul din buclă NU se execută pentru a=15.

Variabile locale in for

variable_for.c

```c
#include <stdio.h>

int main () {

 int a;
 int b = 10;
 printf("b inainte de for %d\n", b);
 for( a = 10; a < 15; a = a + 1 ){
    printf("a = %d\n", a);
    int c = 2;
    a++;
    b = a;
    c = a;
    printf("b in for %d\n", b);
    printf("c in for %d\n", c);
 }
 printf("b dupa for %d\n", b);
 printf("c dupa for %d\n", c);

 return 0;
}
```

https://onlinegdb.com/rknTKSwpX

```
main.c: In function ‘main’:
main.c:20:29: error: ‘c’ undeclared (first use in this function)
  printf("c dupa for %d\n", c);
                            ^
```

```c
#include <stdio.h>

int main () {

 int a;
 int b = 10;
 printf("b inainte de for %d\n", b);

 for( a = 10; a < 15; a = a + 1 ){
   printf("a = %d\n", a);
   int c = 2;
   a++;
   b = a;
   c = a;
   printf("b in for %d\n", b);
   printf("c in for %d\n", c);
 }
 printf("b dupa for %d\n", b);
// printf("c dupa for %d\n", c);
 return 0;
}
```

```
b inainte de for 10
a = 10
b in for 11
c in for 11
a = 12
b in for 13
c in for 13
a = 14
b in for 15
c in for 15
b dupa for 15
```

Bucla while față de bucla do while

do-while.c

```
int main()
{
    int a = 10;

    while (a < 10) {
        printf("while a = %d\n", a);
        a++;
    }

    a = 10;

    do {
        printf("do a = %d\n", a);
        a++;
    } while (a < 10);

    return 0;
}
```

https://onlinegdb.com/B1co2rv6m

```
do a = 10
```

Buclele while si do while obligă la adăugarea instrucțiunilor de incrementare a contorului

Bucla while se execută doar dacă expresia din paranteză este adevarată. Condiția este testată înainte de execuția blocului din interiorul buclei while.

Bucla do while se execută înainte de evaluarea expresiei din paranteză. De aceea, indiferent de condiția de adevăr a expresiei din paranteza, bucla se execută cel puțin o dată.

Bucla while față de bucla do while

sugerat de dl. Alexandru Palcau

Bucla while față de bucla for

Bucla for se folosește în situațiile în care avem un număr fix de iterații cunoscut dinainte de execuție. Bucla for se poate întrerupe cu ajutorul instrucțiunii break. De asemenea, valoarea iteratorului se poate influența în interiorul buclei.

Bucla while se folosește în special în situațiile în care numărul de iterații se decide în timpul derulării codului din buclă.

average-while.c

```
int main()
{
   int sum = 0;
   int tmp = 0;
   int cnt = 0;
   float avg;

   while (tmp != -99) {
      printf("suma este: %f\n",avg);
      printf("media este: %f\n",avg);
      printf("Numarul nr. %i\n", cnt);
      scanf ("%i",&tmp);
      cnt = cnt + 1;
      sum = sum + tmp;
      avg = (float)sum / cnt;
   }
   return 0;
}
```

```
suma este: 0
media este: 0.000000
Numarul nr. 1
87
suma este: 87
media este: 43.500000
Numarul nr. 2
67
suma este: 154
media este: 51.333332
Numarul nr. 3
54
suma este: 208
media este: 52.000000
Numarul nr. 4
-99
```

https://onlinegdb.com/HJZSpHwa7

Bucle și decizii

while_if.c

```c
int main()
{
  int sumn = 0, cntn = 0, sump = 0;
  int cntp = 0, cnt = 0;
  int tmp;
  float avgn, avgp;

  while (tmp != -9999) {

     printf("pozitiv count: %i, sum: %i, avg: %f\n",cntp, sump, avgp);
     printf("negativ count: %i, sum: %i, avg: %f\n",cntn, sumn, avgn);

     printf("Numarul %i\n", cnt);
     scanf ("%i",&tmp);
     cnt = cnt + 1;

     if (tmp < 0) {
         cntn++;
         sumn = sumn + tmp;
         avgn = (float)sumn / cntn;
     } else if (tmp > 0) {
         cntp++;
         sump = sump + tmp;
         avgp = (float)sump / cntp;
     }
  }
  return 0;
}
```

https://onlinegdb.com/BJP6pSDaQ

Funcții

Cum să grupăm anumite instrucțiuni în funcții

printf și **scanf** sunt funcții externe (nu fac parte propriu-zis din limbajul C). Sunt implementate în bibliotecile asociate oricărui compilator C, împreună cu alte funcții similare.

Programarea în orice limbaj implică, pe lângă cunoașterea instrucțiunilor limbajului respectiv, cunoașterea funcțiilor disponibile în bibliotecile de sistem precum și a funcțiilor disponibile în alte biblioteci disponibile.

Biblioteca asociată sistemelor Linux (numita glibc) are peste 1600 de funcții.

Utilizarea funcțiilor din bibliotecile externe se face prin folosirea macro-ului **include** care va adăuga la programul curent un fișier special numit header în care se găsesc definițiile funcțiilor respective.

Pentru folosirea funcțiilor printf și scanf a trebuit sa folosim instrucțiunea:

#include <stdio.h> în care stdio.h este headerul.

fragment din fișierul stdio.h

```
/* Write formatted output to STREAM.

  This function is a possible cancellation point
and therefore not
  marked with   THROW.  */
extern int fprintf (FILE *  restrict   stream,
         const char *  restrict   format, ...);
/* Write formatted output to stdout.

  This function is a possible cancellation point
and therefore not
  marked with   THROW.  */
extern int printf (const char *__restrict __format,
...);
/* Write formatted output to S.  */
extern int sprintf (char *  restrict   s,
         const char *__restrict __format, ...)
__THROWNL;
```

Headere ale funcțiilor din glibc.

<assert.h>	<stdatomic.h>
<complex.h>	<stdbool.h>
<ctype.h>	<stddef.h>
<errno.h>	<stdint.h>
<fenv.h>	<stdio.h>
<float.h>	<stdlib.h>
<inttypes.h>	<stdnoreturn.h>
<limits.h>	<string.h>
<locale.h>	<tgmath.h>
<math.h>	<threads.h>
<setjmp.h>	<time.h>
<signal.h>	<uchar.h>
<stdalign.h>	<wchar.h>
<stdarg.h>	<wctype.h>

Headere frecvent utilizate

Acestea sunt disponibile în marea majoritate a compilatoarelor c

complex.h	Funcții pentru aritmetica numerelor complexe
limits.h	Funcții și macro-uri pentru specificarea valorilor minime și maxime ale tipurilor (ex: funcția sizeof())
math.h	Funcții matematice (sqrt, sin, cos)
stdio.h	Funcții legate de input/output
stdarg.h	Funcții care permit procesarea unui număr variabil de argumente trimise unei funcții.
string.h	Funcții care permit procesarea șirurilor de caractere
time.h	Funcții care permit operații cu structuri de tip dată si oră
uchar.h	Funcții care permit operații cu caractere unicode.

math.h

https://en.wikipedia.org/wiki/C_mathematical_functions

În sistemele de operare tip Linux, funcțiile definite în headerul math.h sunt separate într-o altă billiotecă numită libm. În cazul compilării cu compilatorul gcc (Gnu C Compiler), trebuie adăugată biblioteca libm la instrucțiunea de compilare.

```
gcc -Wall sqrt.c -o sqrt -lm
```

sqrt.c

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

int main()
{
    float x = 4.0, y = 2.0;
    float z1, z2;
    z1 = sqrt(x);
    z2 = sqrt(y);
    printf("%f %f\n", z1, z2);
    return 0;
}
```

https://onlinegdb.com/r1vdABvTX

```
2.000000 1.414214
```

sincos.c

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

const double pi= 3.14159265;
int main()
{
    double x = pi/2;
    double y = sin(x);
    double z = cos(x);
    printf("sin pi/2 = %f\n", y);
    printf("cos pi/2 = %f\n", z);
    return 0;
}
```

https://onlinegdb.com/BJQAArDT7

```
sin pi/2 = 1.000000
cos pi/2 = 0.000000
```

Cum definim o funcție?

functii_putere.c

```c
#include <stdio.h>

int putere(int m, int n);

int main()
{
   int a=3, b=5;
   int c = putere(a, b);
   printf("%d la puterea %d este %d\n",a,b,c);
   return 0;
}

int putere(int m, int n)
{
   int i, p=1;
   for (i=1; i<=n; i++) {
       p = p * m;
   }
   return p;
}
```

Funcția trebuie declarată înainte de a fi utilizată

Apelarea functiei

Definirea funcției

https://onlinegdb.com/ByTCZ6hTQ

```
3 la puterea 5 este 243
```

Variabile locale

functii_putere.c

```c
#include <stdio.h>

int putere(int m, int n);

int main()
{
   int a=3, b=5;
   int c = putere(a, b);
   printf("%d la puterea %d este %d\n",a,b,c);
   return 0;
}

int putere(int m, int n)
{
   int i, p=1;
   for (i=1; i<=n; i++) {
       p = p * m;
   }
   return p;
}
```

Variabilele a, b și c sunt definite în interiorul funcției “main” și sunt accesibile doar în interiorul funcției “main”

Variabilele i, p sunt definite în interiorul funcției **putere**. Acestea sunt variabile locale. Ele nu sunt accesibile în afara funcției în care au fost declarate

https://onlinegdb.com/ByTCZ6hTQ

```
3 la puterea 5 este 243
```

Variabile globale

functii_putere_vars.c

```c
#include <stdio.h>

int putere(int m, int n);
int glb = 121;

int main()
{
   int a=3, b=5;
   int c = putere(a, b);
   printf("in functia main: %i\n", glb);
   printf("%d la puterea %d este %d\n",a,b,c);
   return 0;
}

int putere(int m, int n)
{
   int i, p=1;
   printf("in functia putere: %i\n", glb);
   for (i=1; i<=n; i++) {
       p = p * m;
   }
   return p;
}
```

https://onlinegdb.com/B14OWUwpm

Variabilă globală

Variabilă globală accesibilă în funcția “main”

Variabilă globală accesibilă în funcția “putere”

```
in functia putere: 121
in functia main: 121
3 la puterea 5 este 243
```

Modificarea variabilelor globale

functii_putere_vars_2.c

```c
#include <stdio.h>

int putere(int m, int n);
int glb = 121;

int main()
{
   int a=3, b=5;
   printf("main: 1 %i\n", glb);
   int c = putere(a, b);
   printf("main: 2 %i\n", glb);
   glb = 1234;
   printf("main: 3 %i\n", glb);
   printf("%d la puterea %d este %d\n",a,b,c);
   return 0;
}

int putere(int m, int n)
{
   int i, p=1;
   glb = 4321;
   printf("putere: 1 %i\n", glb);
   for (i=1; i<=n; i++) {
       p = p * m;
   }
   return p;
}
```

https://onlinegdb.com/SJabWUDpX

```
main: 1 121
putere: 1 4321
main: 2 4321
main: 3 1234
3 la puterea 5 este 243
```

Variabilele globale pot fi modificate în orice funcție a programului. Odată modificate, celelalte apeluri ale aceleiași variabile vor utiliza valoarea modificată.

Nume de variabile și nume de funcții

main_inside_main.c

```c
#include <stdio.h>
int putere(int m, int n)
{
    int i, main=1;
    for (i=1; i<=n; i++) {
        main = main * m;
    }
    return main;
}

int main()
{
    int main = 3;
    printf("%d\n", main);
    printf("Putere: %d\n", putere(main, 5));
    return 0;
}
```

https://onlinegdb.com/0iWTcNL4J

Numele de funcții pot fi folosite ca variabile (nimeni nu știe de ce am face asta) în condițiile în care ele sunt definite în interiorul funcției respective sau în interiorul altei funcții astfel încât funcțiile să se afle în afara scopului variabilelor.

```
main-inside-main.c: In function 'main':
main-inside-main.c:5:9: warning: 'main' is usually a function [-Wmain]
     int main = 3;
         ^
```

```
3
```

Constante globale

global_const.c

```c
#include <stdio.h>

const double PI=3.14159265;

double aria(double raza)
{
   return PI * raza * raza;
}

double lungime(double raza)
{
   return 2 * PI * raza;
}

int main()
{
   double raza = 12;
   printf("Aria cercului: %f\n", aria(raza));
   printf("Lungimea cercului: %f\n",
lungime(raza));
   return 0;
}
```

Constantă globală

https://onlinegdb.com/S1f7G8wp7

```
Aria cercului: 452.389342
Lungimea cercului: 75.398224
```

Funcțiile trebuie declarate înainte de a fi utilizate

functii_putere.c

```c
#include <stdio.h>

int putere(int m, int n);

int main()
{
   int a=3, b=5;
   int c = putere(a, b);
   printf("%d la puterea %d este
%d\n",a,b,c);
   return 0;
}

int putere(int m, int n)
{
   int i, p=1;
   for (i=1; i<=n; i++) {
       p = p * m;
   }
   return p;
}
```

https://onlinegdb.com/H1A97UPTm

global_const.c

```c
#include <stdio.h>

const double PI=3.14159265;

double aria(double raza)
{
   return PI * raza * raza;
}

double lungime(double raza)
{
   return 2 * PI * raza;
}

int main()
{
   double raza = 12;
   printf("Aria cercului: %f\n",
aria(raza));
   printf("Lungimea cercului: %f\n",
lungime(raza));
   return 0;
}
```

https://onlinegdb.com/H1wP7UvpX

Declarația poate fi însoțită de corpul funcției sau nu. În cazul în care corpul funcției este specificat dincolo de funcțiile în care este apelată, ea poate fi declarată la începutul programului.

Variabile și funcții

variable_funct.c

```c
#include <stdio.h>

int putere(int m, int n);

int main()
{
  int a=3, b=5;
  printf("inainte de apel: a=%d, b=%d\n", a,b);
  int c = putere(a, b);
  printf("%d la puterea %d este %d\n",a,b,c);
  printf("dupa apel: a=%d, b=%d\n", a,b);
  return 0;
}

int putere(int m, int n)
{
  int i, p=1;
  int a = 12;
  int b = 21;
  printf("In functie: a=%d, b=%d\n", a,b);
  for (i=1; i<=n; i++) {
      p = p * m;
  }
  return p;
}
```

https://onlinegdb.com/rJWM48v6X

```
inainte de apel: a=3, b=5
In functie: a=12, b=21
3 la puterea 5 este 243
dupa apel: a=3, b=5
```

Șiruri, matrici

Cum să operăm cu date multiple (șiruri, matrici)

average-while.c

```c
int main()
{
   int sum = 0;
   int tmp = 0;
   int cnt = 0;
   float avg;

   while (tmp != -99) {
      printf("suma este: %f\n",avg);
      printf("media este: %f\n",avg);
      printf("Numarul nr. %i\n", cnt);
      scanf ("%i",&tmp);
      cnt = cnt + 1;
      sum = sum + tmp;
      avg = (float)sum / cnt;
   }
   return 0;
}
```

https://onlinegdb.com/SJbCNUPaX

Datele pe care le introducem de la tastatură, deși sunt folosite la calculul mediei aritmetice, se pierd.

Dacă am dori să stocăm undeva aceste date, cu ceea ce știm până acum, ar trebui să inițializăm un număr suficient de variabile și apoi să citim fiecare variabilă în parte.

Aceste variabile au o serie de lucruri în comun:

- Au același tip (numere întregi)
- Reprezintă acelasi lucru (numere date pentru calculul mediei).

Ar fi util ca aceste numere să poată fi grupate și marcate ca fiind asociate.

Pentru acest tip de problema, limbajele de programare oferă diferite structuri de date care grupează tipurile fundamentale de date în liste. Acestea se numesc **șiruri sau matrici**.

Calculul mediei cu ajutorul unui șir

Definirea unui șir trebuie să conțină tipul de date din care vor fi formate elementele șirului împreună cu numărul de elemente ale acestuia.

Ex: int val[15] va defini un șir de 15 numere întregi.

avg_array.c

```
int main()
{
   int N = 5;
   int val[N];
   int i = 0;
   int sum = 0;
   float avg;
   for ( i = 0; i < N; i++ ) {
     printf("Introduceti al %i-lea numar: ",i);
     scanf("%d", &val[i]);
   }

   for ( i = 0; i < N; i++ ) {
       printf("Al %i-lea numar este: %i\n",i, val[i]);
       sum = sum + val[i];
   }

   avg = (float)sum/N;
   printf("Media este: %f, Suma este %i\n", avg, sum);

   return 0;
}
```

```
Introduceti al 0-lea numar: 21
Introduceti al 1-lea numar: 14
Introduceti al 2-lea numar: 19
Introduceti al 3-lea numar: 32
Introduceti al 4-lea numar: 21
Al 0-lea numar este: 21
Al 1-lea numar este: 14
Al 2-lea numar este: 19
Al 3-lea numar este: 32
Al 4-lea numar este: 21

Media este: 21.400000, Suma
este 107
```

https://onlinegdb.com/HkVerIP6m

Șiruri in C

- Un șir în C este o colecție de date de același tip care sunt stocate în poziții de memorie succesive.
- Un șir poate conține **un singur tip** de variabile
- Un șir poate conține orice tip de variabilă, atâta timp cât t**oți membrii săi sunt la fel**.
- Dimensiunea șirului (numărul de variabile pe care le conține) trebuie **să fie cunoscută** la momentul definirii acestuia.
- Elementele din șir se pot accesa cu ajutorul indexului. sir[0] este primul element, sir[1] al doilea etc.
- Adresa șirului este adresa primului element al acestuia (elementul cu index 0).
- Șirurile pot fi traversate prin folosirea indexului.

```
for ( i = 0; i < N; i++ ) {
  printf("Valoarea lui %i: %d\n",i, val[i]);
}
```

Localizarea în memorie a membrilor unui șir

array_memory_char.c

```c
int main()
{
    int N = 10;
    char val[N];
    int i = 0;
    for ( i = 0; i < N; i++ ) {
      printf("Adresa %i: %p\n",i, &val[i]);
    }
    return 0;
}
```

https://onlinegdb.com/H1bNdTSa7

```
Adresa 0: 0x7ffdfd5e8610
Adresa 1: 0x7ffdfd5e8611
Adresa 2: 0x7ffdfd5e8612
Adresa 3: 0x7ffdfd5e8613
Adresa 4: 0x7ffdfd5e8614
Adresa 5: 0x7ffdfd5e8615
Adresa 6: 0x7ffdfd5e8616
Adresa 7: 0x7ffdfd5e8617
Adresa 8: 0x7ffdfd5e8618
Adresa 9: 0x7ffdfd5e8619
```

Diferența dintre două valori succesive: 1 octet

array_memory_short.c

```c
int main()
{
    int N = 10;
    short val[N];
    int i = 0;
    for ( i = 0; i < N; i++ ) {
      printf("Adresa %i: %p\n",i, &val[i]);
    }
    return 0;
}
```

https://onlinegdb.com/Syxia_TrTX

```
Adresa 0: 0x7ffecf392a50
Adresa 1: 0x7ffecf392a52
Adresa 2: 0x7ffecf392a54
Adresa 3: 0x7ffecf392a56
Adresa 4: 0x7ffecf392a58
Adresa 5: 0x7ffecf392a5a
Adresa 6: 0x7ffecf392a5c
Adresa 7: 0x7ffecf392a5e
Adresa 8: 0x7ffecf392a60
Adresa 9: 0x7ffecf392a62
```

Diferența dintre două valori succesive: 2 octeți

În funcție de dimensiunea membrilor șirului, locația de memorie a fiecărui membru arată continuitatea lor în memorie.

Localizarea în memorie a membrilor unui șir

array_memory_int.c

```c
int main()
{
    int N = 10;
    int val[N];
    int i = 0;
    for ( i = 0; i < N; i++ ) {
     printf("Adresa %i: %p\n",i, &val[i]);
    }
    return 0;
}
```

https://onlinegdb.com/rymOP6rTQ

```
Adresa 0: 0x7ffc458a2a70
Adresa 1: 0x7ffc458a2a74
Adresa 2: 0x7ffc458a2a78
Adresa 3: 0x7ffc458a2a7c
Adresa 4: 0x7ffc458a2a80
Adresa 5: 0x7ffc458a2a84
Adresa 6: 0x7ffc458a2a88
Adresa 7: 0x7ffc458a2a8c
Adresa 8: 0x7ffc458a2a90
Adresa 9: 0x7ffc458a2a94
```

Diferența dintre două valori succesive: 4 octeți

array_memory_long.c

```c
int main()
{
    int N = 10;
    long val[N];
    int i = 0;
    for ( i = 0; i < N; i++ ) {
     printf("Adresa %i: %p\n",i, &val[i]);
    }
    return 0;
}
```

https://onlinegdb.com/SJrQP6BaQ

```
Adresa 0: 0x7fff5161c040
Adresa 1: 0x7fff5161c048
Adresa 2: 0x7fff5161c050
Adresa 3: 0x7fff5161c058
Adresa 4: 0x7fff5161c060
Adresa 5: 0x7fff5161c068
Adresa 6: 0x7fff5161c070
Adresa 7: 0x7fff5161c078
Adresa 8: 0x7fff5161c080
Adresa 9: 0x7fff5161c088
```

Diferența dintre două valori succesive: 8 octeți

În funcție de dimensiunea membrilor șirului, locația de memorie a fiecărui membru arată continuitatea lor în memorie.

Inițializarea șirurilor

array_init.c

```c
int main()
{
    int N = 10;
    int a[] = { 0 };
    int b[5] = { 0 };
    int c[] = {};
    int d[5] = {};
    int e[] = { 5, 5, 5, 5, 5 };
    int f[5] = { 5, 5, 5};
    int g[N];

    printf ("a: %ld\n", sizeof(a)/sizeof(a[0]));
    printf ("b: %ld\n", sizeof(b)/sizeof(b[0]));
    printf ("c: %ld\n", sizeof(c)/sizeof(c[0]));
    printf ("d: %ld\n", sizeof(d)/sizeof(d[0]));
    printf ("e: %ld\n", sizeof(e)/sizeof(e[0]));
    printf ("f: %ld\n", sizeof(f)/sizeof(f[0]));
    printf ("g: %ld\n", sizeof(g)/sizeof(g[0]));
    return 0;
}
```

https://onlinegdb.com/Hk_Aw8DaQ

```
a: 1
b: 5
c: 0
d: 5
e: 5
f: 5
g: 10
```

Un sir cu un singur membru, care este 0.

Declararea unui șir cu 5 elemente inițializate la 0.

Declarare fără dimensiune și fără inițializare. Nu există nici o sursă de informații despre dimensiunea șirului, va fi construit un șir cu 0 elemente

Declarare cu inițializare completă, fără dimensiune.

Declarare cu inițializare parțială (celelalte elemente vor fi 0).

Declarare șir cu dimensiune variabilă fără inițializare.

Inițializarea șirurilor

Compilatorul nu poate inițializa șiruri care au dimensiuni variabile.

```
#include <stdio.h>

int main()
{
   int N = 10;
   int g[N] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   printf ("g: %ld\n", sizeof(g)/sizeof(g[0]));
   return 0;
}
```

```
array_init_wr.c: In function 'main':
array_init_wr.c:8:5: error: variable-sized object may not be initialized
    int g[N] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    ^
```

```
#include <stdio.h>
#define N 10

int main()
{
   int g[N] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   printf ("g: %ld\n", sizeof(g)/sizeof(g[0]));
   return 0;
}
```

```
a: 1
b: 5
c: 0
d: 5
e: 5
f: 5
g: 10
```

Ce se întâmplă dacă nu știm dimensiunea?

- Definim șirul suficient de mare pentru a fi acoperitor în toate cazurile.
- Utilizăm alocarea dinamică a memoriei și pointeri.

array_large.c

```c
#include<stdio.h>

#define NR_S 200 /*dimensiune suficient de mare a sirului*/

int main()
{
   int note[NR_S],max,i,studenti,suma=0;
   printf("Numarul de studenti (max %d): ", NR_S);
   scanf("%d", &studenti);
   printf("Introduceti notele\n");
   for(i=0; i < studenti; i++)
   {
       scanf("%d", &note[i]);
   }
   max = 0;
   for(i=0; i< studenti; i++)
   {
      suma = suma + note[i];
      if (note[i] > max)
      {
         max = note[i];
      }
   }
   printf("Cea mai mare nota este %d\n", max);
   printf("Suma este %d\n", suma);
   printf("Media este %f\n", (float)suma/studenti);
   return 0;
}
```

https://onlinegdb.com/HyhlrTBam

```
Numarul de studenti (max 200): 10
Introduceti notele
8
7
6
9
10
10
7
8
9
10
Cea mai mare nota este 10
Suma este 84
Media este 8.400000
```

Trimiterea șirurilor ca parametri ai funcțiilor

array_function_declare_empty.c

```c
#include <stdio.h>


void printint(int st[])
{
   printf ("Lungime in functie : %ld\n", sizeof(st)/sizeof(st[0]));
}

int main()
{
   int c[14] = {};
   printf ("Lungime in main: %ld\n", sizeof(c)/sizeof(c[0]));
   printint(c);
   return 0;
}
```

https://onlinegdb.com/n325XkFON

```
Lungime in main: 14
Lungime in functie : 2
```

?

Chiar dacă pare că trimitem șirul către funcție, în realitate compilatorul înlocuiește șirul cu adresa sa de memorie (pointer). În interiorul funcției obținem dimensiunea pointerului, nu a șirului. De aceea, la transmiterea șirurilor către funcții trebuie trimisă și lungimea acestora.

Trimiterea șirurilor ca parametri ai funcțiilor

array_function_declare_size.c

```
void printint(int st[14])
{
   printf ("Lungime in functie : %ld\n", sizeof(st)/sizeof(st[0]));
}

int main()
{
   int c[14] = {};
   printf ("Lungime in main: %ld\n", sizeof(c)/sizeof(c[0]));
   printint(c);
   return 0;
}
```

https://onlinegdb.com/r1XrNarpm

Parametrul se poate declara cu dimensiunea predefinita, rezultatul e același.

```
Lungime in main: 14
Lungime in functie : 2
```

array_function_declare_pointer.c

```
void printint(int *st)
{
   printf ("Lungime in functie : %ld\n", sizeof(st)/sizeof(st[0]));
}

int main()
{
   int c[14] = {};
   printf ("Lungime in main: %ld\n", sizeof(c)/sizeof(c[0]));
   printint(c);
   return 0;
}
```

https://onlinegdb.com/BJcjEaB6m

Parametrul se poate declara ca pointer, rezultatul e același.

```
Lungime in main: 14
Lungime in functie : 2
```

Trimiterea șirurilor ca parametri ai funcțiilor

array_function_pass_size.c

```c
#include <stdio.h>

void printint(int st[], int size)
{
  int i;
  printf ("Lungime in functie : %ld\n", sizeof(st)/sizeof(st[0]));
  for (i = 0; i < size; i++ ) {
      printf("nr: \%i este %i\n",i, st[i]);
  }
}

int main()
{
  int c[14] = {1,2,3,4,5,6};
  printint(c,10);
  return 0;
}
```

https://onlinegdb.com/S18CQ6rpQ

```
Lungime in functie : 2
nr: 0 este 1
nr: 1 este 2
nr: 2 este 3
nr: 3 este 4
nr: 4 este 5
nr: 5 este 6
nr: 6 este 7
```

Singura variantă pentru a putea enumera șirul în interiorul unei funcții este să trimitem și dimensiunea sa ca parametru suplimentar.

Șiruri multidimensionale (tabele, matrici)

C permite definirea șirurilor cu mai multe dimensiuni. Numărul maxim de dimensiuni este determinat de mărimea tipului de variabilă, de arhitectura sistemului pe care se compilează programul și de alți factori.

```
type name[size1][size2]...[sizeN];
```

```
char arr[3][5] = {};
```

	Coloană 1	Coloană 2	Coloană 3	Coloană 4	Coloană 5
Rând 1	arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]	arr[0][4]
Rând 2	arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]	arr[1][4]
Rând 3	arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]	arr[2][4]

Stocarea în memorie a matricilor

```
int arr[4][3] = {{2,3,5},{6,7,4},{3,9,11},{0,17,10}};
```

arr	0	1	2
0	2	3	5
1	6	7	4
2	3	9	11
3	0	17	10

```
arr[0][0] = 2
arr[0][1] = 3
arr[0][2] = 5
arr[1][0] = 6
arr[1][1] = 7
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 9
arr[2][2] = 11
arr[3][0] = 0
arr[3][1] = 17
arr[3][2] = 10
```

```
arr[0][0] = 2
arr[0][1] = 3
arr[0][2] = 5
arr[0][3] = 6
arr[0][4] = 7
arr[0][5] = 4
arr[0][6] = 3
arr[0][7] = 9
arr[0][8] = 11
arr[0][9] = 0
arr[0][10] = 17
arr[0][11] = 10
```

Surprinzător, dacă tratăm matricea ca pe un șir unidimensional, rezultatele sunt aceleași. În realitate, matricile multidimensionale sunt stocate în memorie tot ca șiruri unidimensionale, cu rândurile stocate unul după celălalt.

	0	1	2	3	4	5	6	7	8	9	10	11
0	2	3	5	6	7	4	3	9	11	0	17	10

De aceea, la inițializarea matricilor nu e nevoie de specificarea primei dimensiuni. Declarația de mai sus este echivalentă cu:

```
int arr[][3] = {{2,3,5},{6,7,4},{3,9,11},{0,17,10}};
```

Stocarea în memorie a matricilor

multi_access.c

```c
#include <stdio.h>

int main()
{
   int arr[4][3] =
{{2,3,5},{6,7,4},{3,9,11},{0,17,10}};
   int i;
   int j;

   printf("---Acces multidimensional----\n");

   for(i=0; i<4;i++) {
       for (j=0; j<3; j++) {
         printf("arr[%i][%i] = %d\n",i,j,arr[i][j]);
       }
   }

   printf("\n--Acces unidimensional-----\n");

   for(i=0; i<12;i++) {
        printf("arr[0][%i] = %d\n",i,arr[0][i]);
   }

   return 0;
}
```

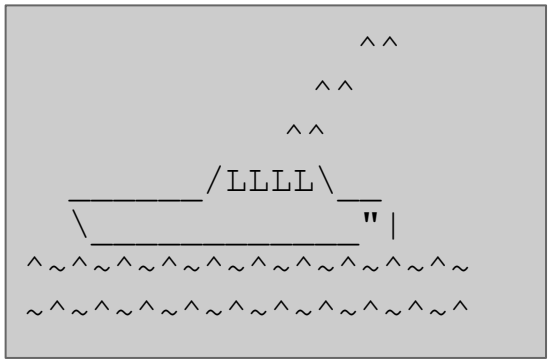
https://onlinegdb.com/B1r4118Tm

```
---Acces multidimensional----
arr[0][0] = 2
arr[0][1] = 3
arr[0][2] = 5
arr[1][0] = 6
arr[1][1] = 7
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 9
arr[2][2] = 11
arr[3][0] = 0
arr[3][1] = 17
arr[3][2] = 10

---Acces unidimensional---
arr[0][0] = 2
arr[0][1] = 3
arr[0][2] = 5
arr[0][3] = 6
arr[0][4] = 7
arr[0][5] = 4
arr[0][6] = 3
arr[0][7] = 9
arr[0][8] = 11
arr[0][9] = 0
arr[0][10] = 17
arr[0][11] = 10
```

Șiruri multidimensionale (tabele, matrici)

```c
#include<stdio.h>

int main()
{

   char arr[7][20] = {};
   int i,j;

   arr[0][15] = 94;
   arr[0][16] = 94;
   arr[1][13] = 94;
   . . . . .
   arr[6][18] = 126;
   arr[6][19] = 94;

   for(i=0; i<=6;i++) {
       for (j=0; j<20; j++) {
            if (arr[i][j] > 0) {
                printf("%c",arr[i][j]);
            } else {
                printf(" ");
            }
       }
       printf("\n");
   }

return 0;
}
```

https://onlinegdb.com/BJjlv8Dpm



“Vizitarea” tuturor membrilor unei matrici se numește și traversarea matricii.

Traversarea unei matrici se face prin utilizarea buclelor repetitive.

În interiorul buclelor se plasează de obicei structuri decizionale.

Stocarea în memorie a matricilor

array_multi_uni.c

```c
#include <stdio.h>

int main()
{
   int arr[2][4][3] = {
    { {2,3,5}, {6,7,4},{3,9,11},{0,17,10} },
    { {22,23,25}, {26,27,24}, {23,29,211},
{20,217,210} }
   };
   int i,j,k;
   printf("---Acces multidimensional-------\n");
   for(i=0; i<2;i++) {
       for (j=0; j<4; j++) {
           for (k=0; k<3; k++) {
               printf("arr[%i][%i][%i] = %d\n",
i,j,k,arr[i][j][k]);
           }
       }
   }
   printf("\n---Acces unidimensional-------\n");
   for(i=0; i<24;i++) {
        printf("arr[0][0][%i] = %d\n",
i,arr[0][0][i]);
   }
   return 0;
}
```

https://onlinegdb.com/ByM0U5Ya7

```
arr[0][0][0] = 2
arr[0][0][1] = 3
arr[0][0][2] = 5
arr[0][1][0] = 6
arr[0][1][1] = 7
arr[0][1][2] = 4
arr[0][2][0] = 3
arr[0][2][1] = 9
arr[0][2][2] = 11
arr[0][3][0] = 0
arr[0][3][1] = 17
arr[0][3][2] = 10
arr[1][0][0] = 22
arr[1][0][1] = 23
arr[1][0][2] = 25
arr[1][1][0] = 26
arr[1][1][1] = 27
arr[1][1][2] = 24
arr[1][2][0] = 23
arr[1][2][1] = 29
arr[1][2][2] = 211
arr[1][3][0] = 20
arr[1][3][1] = 217
arr[1][3][2] = 210
```

```
arr[0][0][0] = 2
arr[0][0][1] = 3
arr[0][0][2] = 5
arr[0][0][3] = 6
arr[0][0][4] = 7
arr[0][0][5] = 4
arr[0][0][6] = 3
arr[0][0][7] = 9
arr[0][0][8] = 11
arr[0][0][9] = 0
arr[0][0][10] = 17
arr[0][0][11] = 10
arr[0][0][12] = 22
arr[0][0][13] = 23
arr[0][0][14] = 25
arr[0][0][15] = 26
arr[0][0][16] = 27
arr[0][0][17] = 24
arr[0][0][18] = 23
arr[0][0][19] = 29
arr[0][0][20] = 211
arr[0][0][21] = 20
arr[0][0][22] = 217
arr[0][0][23] = 210
```

Returnarea unei matrici dintr-o funcție

- Matricile și șirurile care sunt declarate în main și trimise către funcții, sunt trimise ca referințe (pointeri). Dacă acestea sunt modificate în funcții, modificările persistă și după ce funcția se încheie.

- Matricile declarate în interiorul funcțiilor sunt variabile locale - acestea nu persistă după încheierea funcțiilor. Dacă dorim să declarăm o matrice în interiorul unei funcții și să o returnăm, trebuie să folosim o metodă diferită de declarare, folosind funcția **malloc**.

- Declararea unei matrici de dimensiuni mari într-o funcție (adică în stack) poate depăși dimensiunea stackului provocând eroarea “stack overflow”.

- Malloc alocă memoria în alt spațiu (heap) care nu se resetează la ieșirea din funcție. Heap-ul este de obicei semnificativ mai mare decât stackul.

- La părăsirea funcției, se poate returna un pointer către matricea declarata cu malloc si aceasta va fi disponibilă după încheierea funcției.

Returnarea unei matrici dintr-o funcție

insertion_copy.c

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

int main()

{
   int data[] = {12,23,5,7,2,9,11,13,9};
   int *sorted = NULL;
   int num = sizeof(data)/sizeof(data[0]);

   printf("Inainte de sort:\n");
   printstr(data,num);
   sorted = insertion(data, num);
   printf("Dupa sort :\n");
   printstr(data,num);
   printf("Sorted :\n");
   printstr(sorted,num);
   return 0;
}
```

https://onlinegdb.com/SygL1OwpQ

```c
int * insertion(int data[], int size)
{
   int *newArr;
   int i,temp,j;
   newArr=(int*)malloc(size*sizeof(int));
   for (i = 1 ; i <= size - 1; i++)
   {
       newArr[i] = data[i];
   }
   for (i = 1 ; i <= size - 1; i++)
   {
     j = i;
    while ( j > 0 && newArr[j] < newArr[j-1])
       {
           temp = newArr[j];
           newArr[j] = newArr[j-1];
           newArr[j-1] = temp;
           j--;
       }
    }
    return newArr;
}
```

```c
void printstr(int data[], int size)
{
    int i;
    for (i = 0; i <= size - 1; i++)
    {
          printf("%d\n", data[i]);
    }
}
```

Variabile, pointeri și referințe

Universul unui program C

Stack	Spațiul în care sunt stocate variabilele non-statice și apelurile de funcții

memorie distribuită, biblioteci dinamice.

Heap	spațiul disponibil pentru alocare dinamică a memoriei în timpul rulării programului (malloc, calloc)
BSS	Spațiul în care sunt stocate varabilele statice neinițializate (static int x) - care vor fi automat initializate cu 0
Data	Spațiul în care sunt stocate varabilele statice inițializate (static int x = 12)
Text	Spațiul în care se găsește programul (instrucțiunile binare ale acestuia)

Stack-ul crește sau scade la apelul funcțiilor. Fiecare apel de funcție este reprezentat în stack sunt forma unei structuri de date numita “stack frame”. Managementul stack-ului este făcut de sistemul de operare cu ajutorul unor registre ale procesorului.

Heap-ul este variabil, crește (sau scade) în timpul execuției programului în funcție de memoria alocată

Data nu este read-only (sau are cel puțin o secțiune modificabila) - variabilele se pot schimba

De obicei segmentul text este read-only dar poate fi distribuit între mai multe procese

https://gcc.godbolt.org/

gcc.godbolt.org

COMPILER EXPLORER

Add... More

Use conan or vcpkg to manage your C & C++ library dependencies

Sponsors intel. PC-lint

Share Other Policies

C source #1

```c
int square(int d);

int main()
{
  int a = 100;
  int m = square(a);
}

int square (int d) {
  int prd = d * d;
  return prd;
}
```

x86-64 gcc 10.3 (Editor #1, Compiler #1) C

x86-64 gcc 10.3 | Compiler options...

Output... Filter... Libraries + Add new... Add tool...

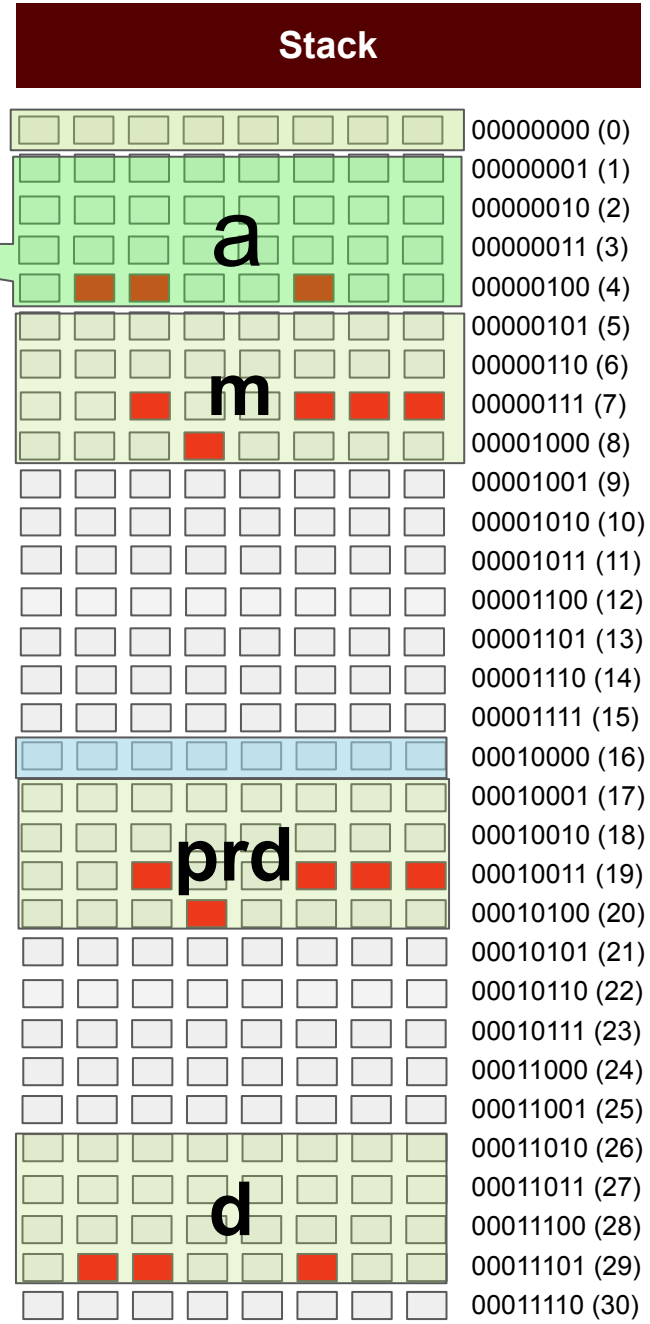
```asm
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 100
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square
        mov     DWORD PTR [rbp-8], eax
        mov     eax, 0
        leave
        ret
square:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     eax, DWORD PTR [rbp-20]
        imul    eax, eax
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

Pentru mai multa distracție cu compilatoarele si assembler

Variabile și referințe

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 100
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square
        mov     DWORD PTR [rbp-8], eax
        mov     eax, 0
        leave
        ret
square:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-10], edi
        mov     eax, DWORD PTR [rbp-10]
        imul    eax, eax
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

```
int square(int d);

int main()
{
  int a = 100;
  int m = square(a);
}

int square (int d) {
  int prd = d * d;
  return prd;
}
```




```
mov DWORD PTR [rbp-4], 100
```

Valoarea 100, cu dimensiunea DWORD se scrie în adresa PTR[rbp-4]. Acesta este un pointer!

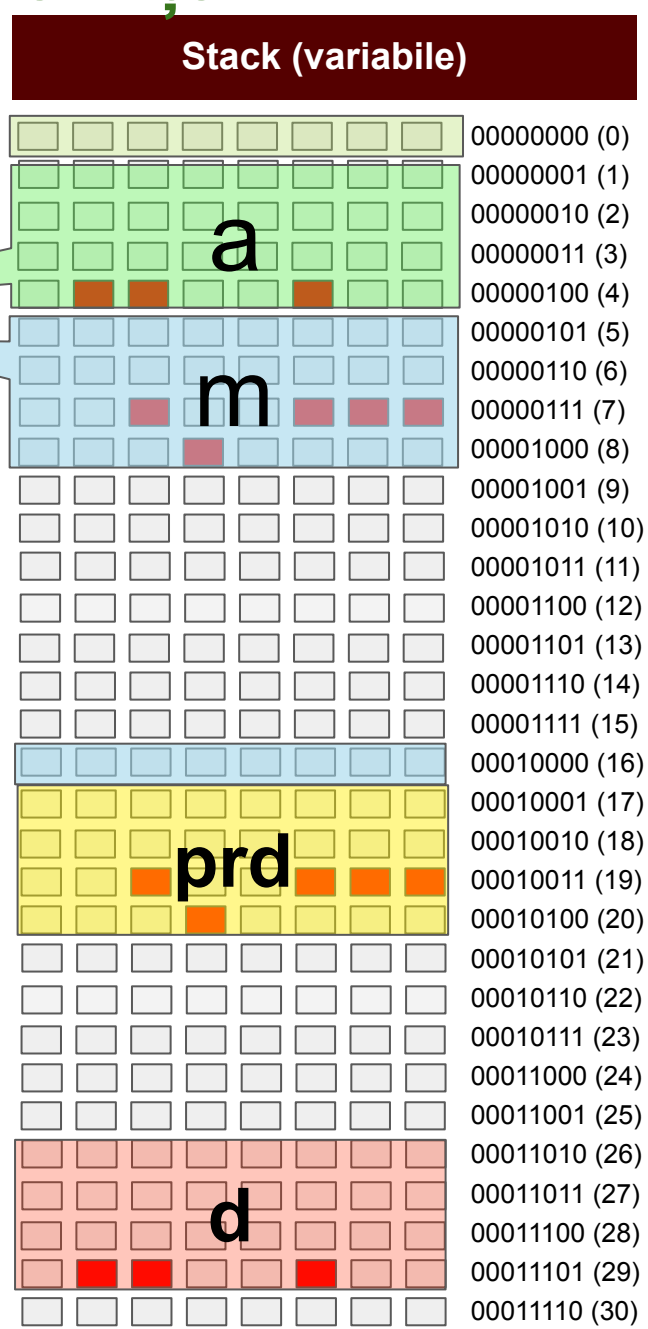
rsp (baza stack pointer-ului)
rbp (baza frame pointer)
eax (acumulator)
edi (destination register)

```
push    rbp
mov     rbp, rsp
```

Se execută la intrarea în orice funcție (inclusiv main) - se adaugă frame-pointerul la stivă (pentru mai târziu) și se corectează frame-pointerului cu baza stivei functiei curente.

Variabile și referințe

```
int square(int d);

int main()
{
  int a = 100;
  int m = square(a);
}

int square (int d) {
  int prd = d * d;
  return prd;
}
```



Referințe

Referințele reprezintă prima adresă de memorie dintre cele în care sunt stocate variabilele.

int a
00000001 (1)
00000010 (2)
00000011 (3)
00000100 (4)

&a = 00000001

int m
00000101 (5)
00000110 (6)
00000111 (7)
00001000 (8)

&m = 00000101

int prd
00010001 (17)
00010010 (18)
00010011 (19)
00010100 (20)

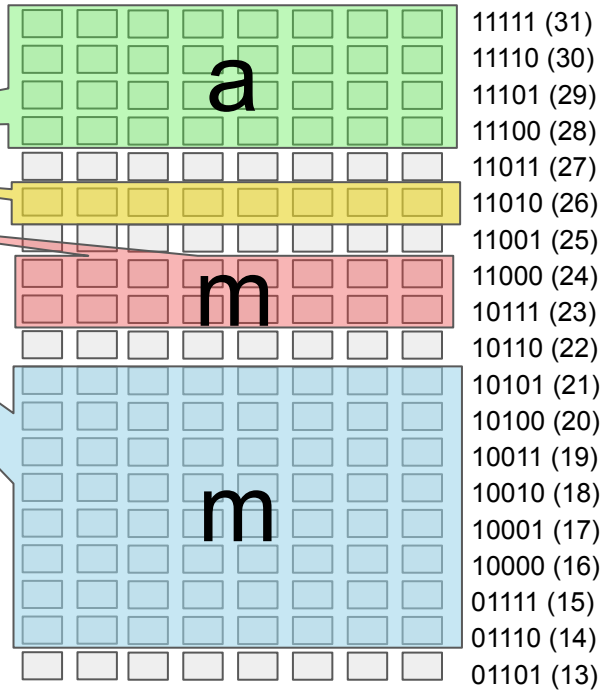
&prd = 00010001

int d
00011010 (26)
00011011 (27)
00011100 (28)
00011101 (29)

&d = 00011010

Variabile și referințe

```
int main()
{
  int a = 100;
  char d = 'b';
  short m = 21;
  long v = 100;
}
```



int a	11110 (30) 11101 (29) 11100 (28) 11011 (27)	**&a = 11110**
char d	11010 (26)	**&d = 11000**
short m	11000 (24) 10111 (23)	**&d = 11000**
long v	10101 (21) 10100 (20) 10011 (19) 10010 (18) 10001 (17) 10000 (16) 01111 (15) 01110 (14)	**&m = 10101**

Exemplu real

variables.c

```
#include <stdio.h>

int main()
{
int a = 100;
printf ("adresa a: %p, lungime %d\n", &a, sizeof(a));
char d = 'b';
printf ("adresa d: %p, lungime %d\n", &d, sizeof(d));
short m = 21;
printf ("adresa m: %p, lungime %d\n", &m, sizeof(m));
long v = 100;
printf ("adresa v: %p, lungime %d\n", &v, sizeof(v));
}
```

```
adresa a: 0x7ffe5fce6084, lungime 4
adresa d: 0x7ffe5fce6081, lungime 1
adresa m: 0x7ffe5fce6082, lungime 2
adresa v: 0x7ffe5fce6088, lungime 8
```

https://onlinegdb.com/SyxGDpYgP_

Referințe

Referințele sunt valori care permit accesarea indirectă a altor date. Alte date pot fi:

- Conținutul altor variabile
- Conținutul unui fișier stocat pe un harddisk
- Conținutul unei anumite zone de memorie
- Conținutul unei alte structuri de date

O referință este diferită de datele respective. De cele mai multe ori, o referință reprezintă **o adresă fizică a unei poziții de memorie** dar sunt situații în care acestea sunt reprezentate altfel. De exemplu, adresa unei pagini internet se numește tot referință.

În sens strict, în programare, referința reprezintă o adresă de memorie care este astfel definită încât compilatorul să poată accesa corect datele respective.

Operația de accesare a datelor din spatele referinței se numește dereferențiere și se face diferit în funcție de limbajul de programare ales.

În C, pentru a accesa datele care se găsesc “în spatele” unei adrese de memorie care se constituie în referință, această adresă trebuie stocată într-un tip special de variabilă numit pointer.

Pointerii se declară împreună cu un tip de variabilă. Astfel, compilatorul știe cât din memorie trebuie să citească în momentul în care este nevoie de dereferențiere.

Variabile și referințe

variables.c

```c
#include <stdio.h>

int main()
{
int a = 100;
printf ("adresa a: %p, lungime %d\n", &a, sizeof(a));
char d = 'b';
printf ("adresa d: %p, lungime %d\n", &d, sizeof(d));
short m = 21;
printf ("adresa m: %p, lungime %d\n", &m, sizeof(m));
long v = 100;
printf ("adresa v: %p, lungime %d\n", &v, sizeof(v));
}
```

https://onlinegdb.com/SJ-dgMiwV

Prima rulare a programului

```
adresa a: 0x7ffe5fce6084, lungime 4
adresa d: 0x7ffe5fce6081, lungime 1
adresa m: 0x7ffe5fce6082, lungime 2
adresa v: 0x7ffe5fce6088, lungime 8
```

A doua rulare a programului

```
adresa a: 0x7ffe5fce6084, lungime 4
adresa d: 0x7ffe5fce6081, lungime 1
adresa m: 0x7ffe5fce6082, lungime 2
adresa v: 0x7ffe5fce6088, lungime 8
```

A treia rulare a programului

```
adresa a: 0x7ffcabb2d1b4, lungime 4
adresa d: 0x7ffcabb2d1b1, lungime 1
adresa m: 0x7ffcabb2d1b2, lungime 2
adresa v: 0x7ffcabb2d1b8, lungime 8
```

De ce sunt adrese diferite?

1. Nici un sistem de operare nu face eforturi pentru a menține baza stack-ului la aceeași adresă.
2. ASLR (address space layout randomization), funcțional în majoritatea sistemelor de operare, “amesteca” cu bună știință adresele pentru a crește securitatea sistemelor.

Consecință: nu putem utiliza adresele de memorie direct în programele noastre. De aceea avem nevoie de un mecanism care să “reînnoiască” adresele de care avem nevoie la fiecare rulare.

Referințe

```
int a = 100;
printf ("adresa a: %p\n", &a);
```

&a se numește “**referință a variabilei a**” și conține adresa de memorie a variabilei a

Referințe în php

```
<?php

$a = 100;
$pa = &$a;
$pa2 = &$pa;

echo "$a\n";
echo "$pa\n";
echo "$pa2\n";
```

```
a = 100
pa = 100
pa2 = 100
```

https://onlinegdb.com/Sk6DBnhUu

Referințe în C++

```
#include <iostream>
using namespace std;

int main () {
   int a = 100;
   int& pa = a;
   int& pa2 = pa;

   cout << "a = " << a << endl;
   cout << "pa = " << pa << endl;
   cout << "pa2 = " << pa2 << endl;
}
```

```
a = 100
pa = 100
pa2 = 100
```

https://onlinegdb.com/Hy8W532Uu

Referințe și pointeri

```
int a = 100;
printf ("adresa a: %p\n", &a);
```

&a se numește “**referință a variabilei a**” și conține adresa de memorie a variabilei a

Putem stoca această adresă de memorie într-un tip special de variabilă, numită “pointer”. Pointer-ul se definește cu ajutorul caracterului * și **trebuie să includă tipul variabilei** către care “punctează”.

```
int *pa = &a;
```

pointer1.c

```
#include <stdio.h>

int main()
{
  int a = 100;
  printf ("a = %d\n", a);
  printf ("adresa a: %p\n", &a);

  int *pa = &a;
  printf("*pa = %p\n", pa) ;
  *pa = 120;
  printf ("a = %d\n", a);
}
```

https://onlinegdb.com/Bkbw-h2Lu

Rezultatul

```
a = 100
adresa a: 0x7ffefdb4f104

*pa = 0x7ffefdb4f104

a = 120
```

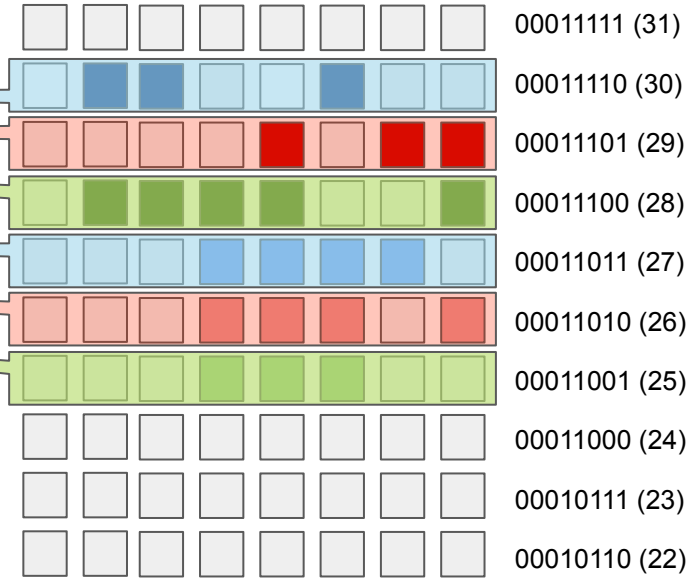
Pointer către char

pointer_char.c

```c
#include <stdio.h>

int main()
{
char a = 100;
char b = 11;
char c = 121;

char *ap = &a;
char *bp = &b;
char *cp = &c;

printf ("adresa a: %p\n", &a);
printf ("adresa b: %p\n", &b);
printf ("adresa c: %p\n", &c);
printf("------------\n");

printf ("valoarea este: %d\n",*ap);
printf ("valoarea este: %d\n",*bp);
printf ("valoarea este: %d\n",*cp);
printf("------------\n");
ap++;
printf ("valoarea este: %d\n",*ap);
ap++;
printf ("valoarea este: %d\n",*ap);
}
```

https://onlinegdb.com/rk7B5o2I_

Conținutul memoriei



01100100 = 100 (a)

00001011 = 11 (b)

01111001 = 121 (c)

00011110 (adresa a)

00011101 (adresa b)

00011100 (adresa c)

```
adresa a: 0x7fffd2c0c425
adresa d: 0x7fffd2c0c426
adresa d: 0x7fffd2c0c427
------------
valoarea este: 100
valoarea este: 11
valoarea este: 121
------------
valoarea este: 11
valoarea este: 121
```

Diferență de 1 octet

Pointerul “sare” din octet în octet când este incrementat.

Pointer către short

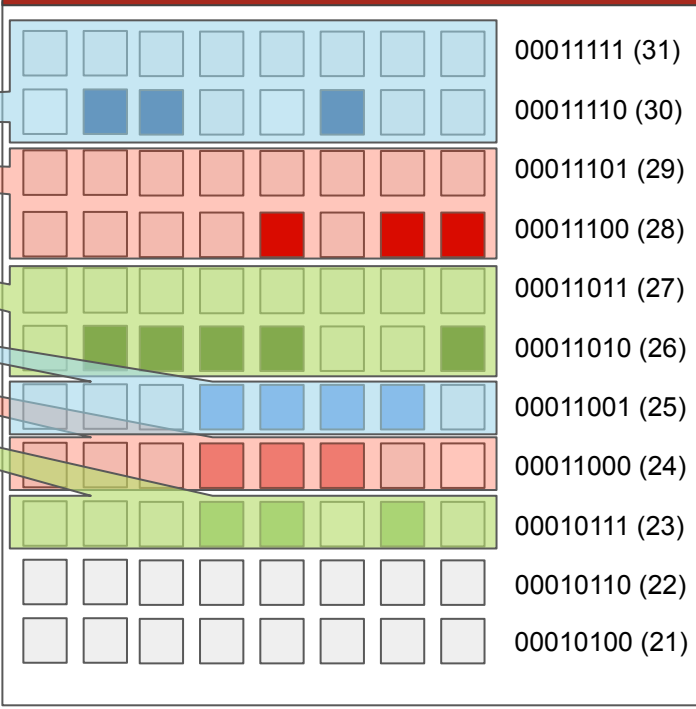
pointer_short.c

```c
#include <stdio.h>

int main()
{
short a = 100;
short b = 11;
short c = 121;

short *ap = &a;
short *bp = &b;
short *cp = &c;

printf ("adresa a: %p\n", &a);
printf ("adresa b: %p\n", &b);
printf ("adresa c: %p\n", &c);
printf("------------\n");

printf ("valoarea este: %d\n",*ap);
printf ("valoarea este: %d\n",*bp);
printf ("valoarea este: %d\n",*cp);
printf("------------\n");
ap++;
printf ("valoarea este: %d\n",*ap);
ap++;
printf ("valoarea este: %d\n",*ap);
}
```

https://onlinegdb.com/rJQez628u

Pointerul “sare” corect din 2 în 2 octeți când este incrementat.

Conținutul memoriei




```
00000000
01100100 = 100 (a)

00000000
00001011 = 11 (b)

00000000
01111001 = 121 (c)

00011110 (adresa a)

00011101 (adresa b)

00011100 (adresa c)
```

```
adresa a: 0x7fff773b69e2
adresa d: 0x7fff773b69e4
adresa d: 0x7fff773b69e6
------------
valoarea este: 100
valoarea este: 11
valoarea este: 121
------------
valoarea este: 11
valoarea este: 121
```

Diferență de 2 octeți

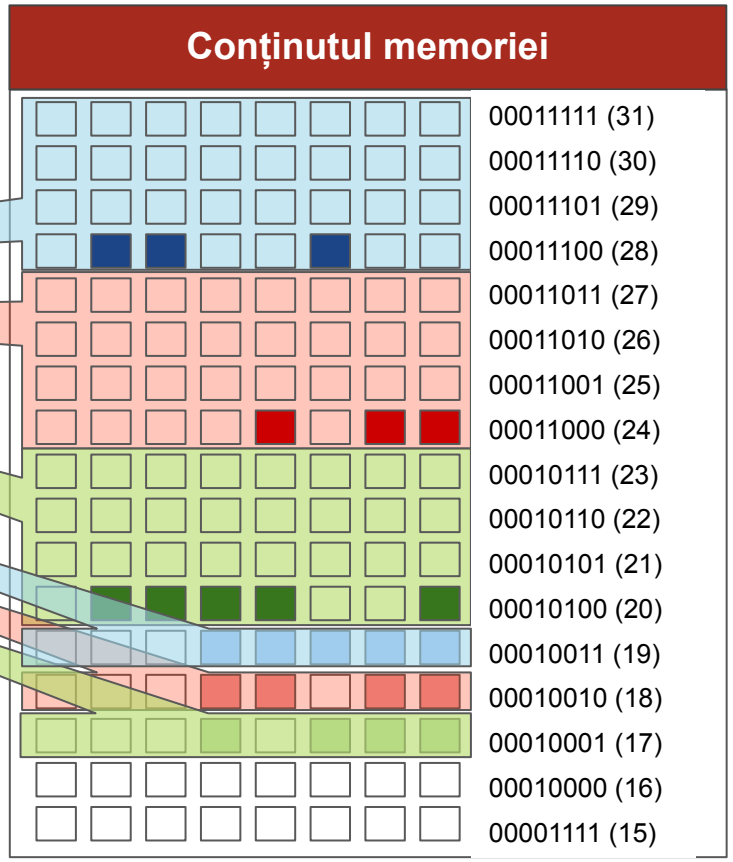
Pointer către int

pointer_int.c

```c
#include <stdio.h>

int main()
{
int a = 100;
int b = 11;
int c = 121;

int *ap = &a;
int *bp = &b;
int *cp = &c;

printf ("adresa a: %p\n", &a);
printf ("adresa b: %p\n", &b);
printf ("adresa c: %p\n", &c);
printf("------------\n");

printf ("valoarea este: %d\n",*ap);
printf ("valoarea este: %d\n",*bp);
printf ("valoarea este: %d\n",*cp);
printf("------------\n");
ap++;
printf ("valoarea este: %d\n",*ap);
ap++;
printf ("valoarea este: %d\n",*ap);
}
```

https://onlinegdb.com/Sk_Kj62Id

Pointerul “sare” corect din 4 în 4 octeți când este incrementat.




```
adresa a: 0x7ffda8a93ccc
adresa d: 0x7ffda8a93cd0
adresa d: 0x7ffda8a93cd4
------------
valoarea este: 100
valoarea este: 11
valoarea este: 121
------------
valoarea este: 11
valoarea este: 121
```

Diferență de 4 octeți

Într-o bibliotecă, există “pointeri” către fiecare dintre cărți. Aceștia conțin “adresa” cărții respective împreună cu numărul de volume.

Adresa cărții conține de obicei informații ca: rândul de rafturi pe care se află, raftul, a câta carte este… etc.

Putem oricând să cerem cartea de la o anumită adresă sau cartea care se află imediat după aceasta sau la distantă de 5 cărți de ea.

Ce este de fapt un pointer?

Un pointer este un tip special de variabilă care stochează adresa de memorie a unei alte variabile împreună cu dimensiunea acesteia.

Un pointer știe exact **poziția la care începe stocarea unei variabile în memorie**

Un pointer știe exact **numărul de octeți utilizat la stocarea variabilelor.** Aceasta este posibil pentru că definirea pointerilor se face împreună cu tipul de variabilă către care acesta trimite.

Dimensiunea propriu-zisă a unui pointer (a variabilei de tip pointer) este aceeași de fiecare dată, indiferent de tipul de variabilă către care trimite.

Un pointer poate fi incrementat sau decrementat, acesta “mutându-se” de la începutul unei variabile la începutul următoarei (sau a celei precedente). Aceasta se numește “**aritmetica pointerilor**”.

Un pointer poate fi utilizat pentru obținerea valorii variabilei către care “punctează”. Această operație se numește **dereferențiere**.

Pointer

```
#include <stdio.h>

int main()
{
   int a;
   int *aptr;
   int b;
   a = 1021;
   aptr = &a;
   b = *aptr;
   printf ("a = %d, aptr = %p, b = %d\n", a,aptr,b);
   return 0;
}
```

https://onlinegdb.com/BkBCd91cN

pointer2.c

Declarăm variabila a, de tip int.

Declarăm variabila aptr, de tip pointer către int.

Declarăm variabila b

Atribuim valoarea 1021 variabilei a

Atribuim referința către variabila a pointerului aptr.

Atribuim valoarea de la adresa către care trimite pointerul aptr variabilei b.

```
a = 1021, aptr = 0x7ffd000aa770, b = 1021
```

Declarația pointerului este întotdeauna însoțită de tipul variabilei către care va “puncta”.

```
<tip> * <nume_variabila>
```

Operatorul * (dereferențiere sau indirection operator) permite accesul datelor aflate în spatele unui pointer. De asemenea, declararea pointerului se face tot prin utilizarea operatorului *

Rolurile asteriscului și ampersandului

```
int a = 1021;

int *aptr;

int b = *aptr;

int c = x * y

aptr = &a;

c = x&y

if (x > 0 && y > 0)
```

*	Specificator pentru declararea unei variabile de tip pointer
*	Operator de dereferențiere
*	Operator pentru operația de înmulțire
&	Specificator al referinței
&	Operație binară AND
&&	Operație logică AND

Declararea pointerilor

Declararea mai multor pointeri nu se poate face pe aceeași linie. In situația următoare:

```
int * p1, p2, p3;
```

doar p1 va fi pointer, celelalte două variabile (p2 și p3) vor fi variabile obișnuite de tip int. Pentru a le declara pe toate trei ca fiind pointeri, trebuie fie scrise una câte una:

```
int * p1;
int * p2;
int * p3;
```

sau plasat operatorul * la începutul fiecărui nume de variabila

```
int *p1, *p2, *p3;
```

[] are o precedență mai mare decât *, din următoarea declarație rezultă o matrice de 10 pointeri de tip int și NU un pointer către matricea ar[10]

```
int * ar[10];
```

Declararea pointerilor

Se pot defini pointeri către pointeri. În declarația următoare var este un pointer către un pointer de tip int.

```
int * * var;
```

Nu contează unde este așezat operatorul * la declararea unui pointer, toate variantele sunt permise:

```
int* p1;
int *p2;
int * p3;
```

Se poate defini un pointer către o constantă. Aceasta declarație interzice modificarea valorii variabilei var prin intermediul pointerului.

var poate să nu fie constantă.

```
const int *ans = &var;
```

Se poate defini un pointer constant către o variabila. *ans va permite modificarea valorii var dar pointerul în sine nu poate fi modificat (nu poate trimite către o altă variabilă)

```
int *const ans = &var;
```

Erori: neinițializarea pointerilor

pointer_neinitializat.c

```c
#include <stdio.h>

int main()
{
   int a;
   int *aptr;
   int b;
   a = 1021;
//    aptr = &a;
   b = *aptr;
   printf ("a = %d, aptr = %p, b = %d\n", a,aptr,b);
   return 0;
}
```

pointer neinițializat.

https://onlinegdb.com/SJ8Lj9yc4

Declararea unui pointer neinițializat nu reprezintă în sine o problemă, problema apare la **încercarea de dereferențiere** a acestuia.

În lipsa inițializării, la încărcarea programului în memorie (memoria nu se golește, se suprascrie), pointerul va găsi niște biți pe care **îi va interpreta ca pe o adresă**. La dereferențiere va încerca să acceseze o adresă **care nu există**, declanșând erori de tipul segmentation fault sau general protection fault.

```
In function ‘main’:
10:7: warning: ‘aptr’ is used uninitialized in this function
[-Wuninitialized]
        b = *aptr;
        ~~^~~~~~~
```

Compilarea emite o atenționare (dacă se folosesc opțiunile corespunzătoare).

```
Segmentation fault (core dumped)
```

La execuție… :)

Erori: Atribuirea unei valori greșite

pointer_gresit_atribuit.c

```c
#include <stdio.h>

int main()
{
   int a;
   int *aptr;
   int b;
   a = 1021;
   aptr = a;
   b = *aptr;
   printf ("a = %d, aptr = %p, b = %d\n", a,aptr,b);
   return 0;
}
```

în loc de aptr = &a;

https://onlinegdb.com/S1Gzi5y5V

Atribuirea unei valori greșite (care nu constituie un spațiu de memorie corect) unui pointer determină de asemenea o eroare, în momentul apelării pointerului.

La rulare, a va fi interpretat ca o adresă de memorie (ceea ce nu este).

La dereferențiere va încerca să acceseze o adresă care nu există, declanșând erori de tipul segmentation fault sau general protection fault.

```
In function 'main':
9:10 warning: assignment to 'int *' from 'int' makes pointer from
integer without a cast [-Wint-conversion]
        aptr = a;
             ^
```

Compilarea emite o atenționare.

```
Segmentation fault (core dumped)
```

La execuție… :)

Erori: Atribuirea unei valori neinițializate

pointer_val_neinitializat.c

```c
#include <stdio.h>

int main()
{
   int a;
   int *aptr;
   int b;
//    a = 1021;
   aptr = &a;
   b = *aptr;
   printf ("a = %d, aptr = %p, b = %d\n", a,aptr,b);
   return 0;
}
```

variabila a rămâne neinițializată

https://onlinegdb.com/BJtR95k94

Atribuirea unei valori neinițializate nu determină o eroare fatală la execuția programului. Dar, programul va opera cu valori greșite, fie numere negative foarte mari , fie 0 în anumite compilatoare

```
In function 'main':
11:5: warning: 'a' is used uninitialized in this function
[-Wuninitialized]
        printf ("a = %d, aptr = %p, b = %d\n", a,aptr,b);
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Compilarea emite o atenționare dar doar cu privire la faptul că variabila este neinițializată.

```
a = -2061479856, aptr = 0x7ffe2c5a3548, b = -2061479856
```

La execuție se obțin valori absurde.

Atenție la pointeri

Atentionare	Mozilla 1.0 1998 (C++) 540613 linii de cod	Apache1.3 - 98 (C) 48393 linii de cod
Using uninitialized memory	26.14	45.00
Dereferencing uninitialized pointer	1.73	0.00
Dereferencing NULL pointer	58.93	50.00
Dereferencing invalid pointer	0.00	5.00
Dereferencing pointer to freed memory	1.98	0.00
Leaking memory	9.75	0.00
Leaking a resource (such as a file)	0.09	0.00
Returning pointer to local stack variable	0.52	0.00
Returning pointer to freed memory	0.09	0.00
Illegal value passed to function	0.43	0.00
Divide by zero	0.34	0.00

A static analyzer for finding dynamic programming errors
By:Bush, WR (Bush, WR); Pincus, JD (Pincus, JD); Sielaff, DJ (Sielaff, DJ)
SOFTWARE-PRACTICE & EXPERIENCE
Volume: 30, Issue: 7, Pages: 775-802

Cu culoarea roșie sunt erori legate de pointere. Valorile sunt procentuale.

NULL pointer

```
int *aptr = NULL;
```

Pointerul NULL este un pointer care trimite către adresa de memorie 0. În majoritatea cazurilor, aceasta adresă este goală.

Încercarea de dereferențiere a unui pointer de tip NULL va duce la o eroare de tip segmentation fault. Dar, un pointer de tip NULL poate fi testat și în majoritatea compilatoarelor, inițializarea pointerilor cu NULL și testarea lor înainte de orice tip de acces reprezintă un mod eficient de a evita erorile.

pointer generic (void pointer)

Un pointer generic este un pointer al cărui tip nu este cunoscut. Este capabil să primească orice fel de adresă fără erori (evident, dereferențierea nu e posibilă - pointerii generici nu pot fi dereferențiați). Practic, un pointer generic nu conține informații despre mărimea variabilei către care trimite. Programatorul va trebui să știe care este dimensiunea memoriei disponibile.

Sunt utilizați de exemplu în funcțiile de alocare a memoriei: acestea trebuie să returneze un pointer (altfel memoria alocată nu va putea fi accesată):

```
exemplu: int * newArr=(int*)malloc(size*sizeof(int));
```

constructia (int*) inaintea funcției malloc convertește pointerul returnat de malloc într-un pointer de tip int.
În multe situații se poate scrie și: int * `newArr = malloc(size*sizeof(int));` fără conversie: multe compilatoare vor converti automat pointerii generici la tipul corect. Utilizarea conversiei în această situație are avantaje și dezavantaje:

https://en.wikipedia.org/wiki/C_dynamic_memory_allocation#Advantages_to_casting

Pointeri și referințe

Referințele sunt valori care permit accesarea indirectă a altor date. Alte date pot fi:

- Conținutul altor variabile
- Conținutul unui fișier stocat pe un harddisk
- Conținutul unei anumite zone de memorie
- Conținutul unei alte structuri de date

Pointerii în C și C++ reprezintă o implementare particulară a referințelor. Alte limbaje de programare utilizează alte feluri de implementări. În C++, conceptul de referință este utilizat pentru a descrie un pointer care nu poate fi modificat, odată declarat și legat de o altă variabilă acesta nu mai poate fi legat de o alta (ca un pointer constant). De asemenea, referințele în C++ nu mai au nevoie de operatorul de dereferențiere

references1.cpp

```
#include <iostream>

using namespace std;

int main () {
 // declararea variabilelor
 int    a = 4;
 double b = 11.21;
 // declararea referintelor
 int&    ra = a;
 double& rb = b;

 cout << "a = " << a << endl;
 cout << "ra : " << ra  << endl;
 cout << "b = " << b << endl;
 cout << "rb = " << rb  << endl;
 return 0;
}
```

https://onlinegdb.com/rJFGjE4uE

- Referințele în C++ permit transferul datelor între funcții.
- Referințele în C++ nu pot avea referințe către ele (în C se pot declara pointeri către pointeri).
- Referințele în C++ nu pot fi nule.
- Referințele în C++ nu pot fi utilizate în operații aritmetice.

```
a = 4
ra : 4
b = 11.21
rb = 11.21
```

Funcții “impure”

O funcție “pură” are următoarele proprietăți:

- Valoarea returnată este de fiecare dată aceeași dacă argumentele sunt aceleași
- Execuția funcției nu are efecte secundare (modificarea altor date cu excepția valorii returnate)

https://onlinegdb.com/SywgbLaUO

```c
#include <stdio.h>

void smfunc(int a) {
   a = 121;
   printf ("In interior: %d\n", a);
}

int main()
{
   int a = 100;
   printf("Inainte: %d\n", a);
   smfunc(a);
   printf("Dupa: %d\n", a);
}
```

Funcția primește ca argument o valoare

```
Inainte: 100
In interior: 121
Dupa: 100
```

impure_nopointer.c

https://onlinegdb.com/BJP-GITU_

```c
#include <stdio.h>

void smfunc(int * a) {
   *a = 121;
   printf ("In interior: %d\n", *a);
}

int main()
{
   int a = 100;
   printf("Inainte: %d\n", a);
   smfunc(&a);
   printf("Dupa: %d\n", a);
}
```

Funcția primește ca argument un pointer

```
Inainte: 100
In interior: 121
Dupa: 121
```

impure_pointer.c

Funcții “impure” - swap

https://onlinegdb.com/H1NMuclw_

```c
#include <stdio.h>

void swap(int *v1,int *v2)
{
   int t;
    t   = *v1;
   *v1   = *v2;
   *v2   =  t;
}
int main()
{
   int a = 100, b=121;

   printf("Inainte: a = %d, b= %d\n", a,b);
   swap(&a, &b);
   printf("Dupa: a = %d, b= %d\n", a,b);
}
```

swap.c

```
Inainte: a = 100, b= 121
Dupa: a = 121, b= 100
```

Funcția primește pointeri către variabilele definite în main.

t este o variabilă temporară folosită pentru a stoca una dintre valori

Se stochează valoarea către care trimite *v1 în t

Se stochează valoarea către care trimite *v2 în *v1 (adresa lui v1)

Se stochează valoarea din t în *v2 (spațiul de memorie al lui v2)

Funcția a manevrat direct variabile definite și stocate în frame-ul funcției părinte, cu ajutorul referințelor și pointerilor.

Pointeri și matrici

array_decay1.c

```c
#include <stdio.h>

int main () {
   int n[] = { 5, 20, 29, 32, 63};

   int vsize = sizeof(n)/sizeof(n[0]);

   printf ("vsize: %d\n", vsize );
}
```

https://onlinegdb.com/HJtE2GC8d

```
vsize: 5
```

La definirea unui tablou, se poate obține dimensiunea acestuia în octeți. Folosind construcția:

```c
sizeof(n)/sizeof(n[0]
```

se poate obține numărul de elemente din tablou.

La trimiterea unui tablou ca argument al unei funcții, acesta pierde informația dimensională pentru că este automat transformat într-un pointer constant.

array_decay2.c

```c
#include <stdio.h>

void functie(int arr[]) {
   int vsize = sizeof(arr)/sizeof(arr[0]);
   printf("vsize in functie, %d\n", vsize);
}

int main () {
   int n[] = { 5, 20, 29, 32, 63 };

   int vsize = sizeof(n)/sizeof(n[0]);

   printf ("vsize in main: %d\n", vsize );

   functie(n);
}
```

https://onlinegdb.com/ryNg9qePu

```
main.c:4:23: warning: ‘sizeof’ on array
function parameter ‘arr’ will return size of
‘int *’ [-Wsizeof-array-argument]
main.c:3:18: note: declared here
vsize in main: 5
vsize in functie, 2
```

Acest comportament se numește “array to pointer decay”. Matricile in C sunt ne-atribuibile (nu putem scrie int d[] = n, ci trebuie sa scriem int * d = n, ceea ce va copia în d doar pointerul “decăzut” al matricii n

Pointeri și matrici

```
int arr[4] = {2,23,5,11};
int *p = arr;
```

Elementele array-ului pot fi accesate prin aritmetica pointerului p.

chiar și "arr" poate fi tratat ca un pointer

	Adresa HEX	Adresa zecimal
2	0x7ffd65f990e0	140726314307808
23	0x7ffd65f990e4	140726314307812
5	0x7ffd65f990e8	140726314307816
11	0x7ffd65f990ec	140726314307820

pointer_array.c

```
printf("adresele primului membru: \n");
printf("arr       este: %p\n", arr);
printf("&(arr[0]) este: %p\n", &(arr[0]));
printf("p         este: %p\n\n", p);

printf("adresele celui de-al doilea membru: \n");
printf("arr+1     este: %p\n", arr+1);
printf("&(arr[1]) este: %p\n", &(arr[1]));
printf("p + 1     este: %p\n\n", p+1);

printf("adresele celui de-al treilea membru: \n");
printf("arr+2     este: %p\n", arr+2);
printf("&(arr[2]) este: %p\n", &(arr[2]));
printf("arr       este: %p\n\n", p+2);

printf("adresele celui de-al patrulea membru: \n");
printf("arr+3     este: %p\n", arr+3);
printf("&(arr[3]) este: %p\n", &(arr[3]));
printf("p + 3     este: %p\n\n", p+3);
```

```
adresele primului membru:
arr       este: 0x7ffd65f990e0
&(arr[0]) este: 0x7ffd65f990e0
p         este: 0x7ffd65f990e0

adresele celui de-al doilea membru:
arr+1     este: 0x7ffd65f990e4
&(arr[1]) este: 0x7ffd65f990e4
p + 1     este: 0x7ffd65f990e4

adresele celui de-al treilea membru:
arr+2     este: 0x7ffd65f990e8
&(arr[1]) este: 0x7ffd65f990e8
arr       este: 0x7ffd65f990e8

adresele celui de-al patrulea
membru:
arr+3     este: 0x7ffd65f990ec
&(arr[3]) este: 0x7ffd65f990ec
p + 3     este: 0x7ffd65f990ec
```

https://onlinegdb.com/SkBT1kzPd

Pointeri și matrici

```
int arr[4] = {2,23,5,11};
int *p = arr;
```

Elementele array-ului pot fi accesate prin aritmetica pointerului p.

chiar și "arr" poate fi tratat ca un pointer

	Adresa HEX	Adresa zecimal
2	0x7ffd65f990e0	140726314307808
23	0x7ffd65f990e4	140726314307812
5	0x7ffd65f990e8	140726314307816
11	0x7ffd65f990ec	140726314307820

Adunarea cu 1 în cazul unui pointer de tip int (4 octeți) va duce la următoarea adresă de număr întreg. Adică, va aduna la adresa curentă 4 octeți.

Adunarea cu 1 în cazul oricărui pointer va duce către adresa următoarei variabile reale, adunând fix atâția octeți cât sunt necesari în funcție de tipul pointerului.

Astfel, în cazul unui pointer de tip char, adunarea cu 1 va adăuga un octet la adresă, în cazul unui short va adăuga doi octeți, în cazul unui double 8 octeți (dependent, evident de arhitectura și de dimensiunea tipurilor pe sistemul pe care se compilează programul

pointer_array1.c

https://onlinegdb.com/SylW0qlDO

```
adresele celui de-al doilea membru:

arr+1      este: 0x7ffd65f990e4
&(arr[1]) este: 0x7ffd65f990e4
p + 1      este: 0x7ffd65f990e4
```

p și arr sunt pointeri de tip int, deci arr + 1 este același lucru cu p + 1.

în cazul folosirii parantezelor pătrate, compilatorul face dereferențierea automata, arr[1] va fi 23 iar pentru a obține adresa este nevoie de operatorul &

Pointeri și matrici - accesul la valori

```
int arr[4] = {2,23,5,11};
int *p = arr;
```

Elementele array-ului pot fi accesate prin aritmetica pointerului p.

chiar și "arr" poate fi tratat ca un pointer

	Adresa HEX	Adresa zecimal
2	0x7ffd65f990e0	140726314307808
23	0x7ffd65f990e4	140726314307812
5	0x7ffd65f990e8	140726314307816
11	0x7ffd65f990ec	140726314307820

pointer_array.c

```
printf("--------------valorile---------- \n");
printf("valorile primului membru: \n");
printf("arr[0]     este: %d\n", arr[0]);
printf("*arr       este: %d\n", *arr);
printf("*p         este: %d\n", *p);

printf("valorile celui de-al doilea membru: \n");
printf("arr[1]     este: %d\n", arr[1]);
printf("*(arr+1)   este: %d\n", *(arr+1));
printf("*(p+1)     este: %d\n", *(p+1));

printf("valorile celui de-al treilea membru: \n");
printf("arr[2]     este: %d\n", arr[2]);
printf("*(arr+2)   este: %d\n", *(arr+2));
printf("*(p+2)     este: %d\n", *(p+2));

printf("valorile celui de-al patrulea membru: \n");
printf("arr[3]     este: %d\n", arr[3]);
printf("*(arr+3)   este: %d\n", *(arr+3));
printf("*(p+3)     este: %d\n", *(p+3));
```

https://onlinegdb.com/SkBT1kzPd

```
--------------valorile----------
valorile primului membru:
arr[0]     este: 2
*arr       este: 2
*p         este: 2

valorile celui de-al doilea membru:
arr[1]     este: 23
*(arr+1)   este: 23
*(p+1)     este: 23

valorile celui de-al treilea membru:
arr[1]     este: 5
*(arr+1)   este: 5
*(p+1)     este: 5

valorile celui de-al patrulea membru:
arr[1]     este: 11
*(arr+1)   este: 11
*(p+1)     este: 11
```

Pointeri și matrici - accesul la valori

```
int arr[4] = {2,23,5,11};
int *p = arr;
```

Elementele array-ului pot fi accesate prin aritmetica pointerului p.

chiar și "arr" poate fi tratat ca un pointer

2	0x7ffd65f990e0	140726314307808
23	0x7ffd65f990e4	140726314307812
5	0x7ffd65f990e8	140726314307816
11	0x7ffd65f990ec	140726314307820

*(arr + 1) adună mai întâi 1 la pointerul arr (deci adună 4 octeți la adresa stocată în pointerul arr ajungând la adresa celui de-al doilea membru al matricii. Aplicând operatorul de dereferențiere pe acea adresă obținem valoarea stocată acolo (23)

arr[1] accesează cel de-al doilea element al matricii (23) - pentru compilator, arr[1] = *((arr)+(1))

!! De aceea indexul matricilor începe de la 0 in C - este un pointer la care nu se adaugă nimic. !!

*(p+1) adună mai întâi 1 la pointerul p (deci adună 4 octeți la adresa stocată în pointerul p ajungând la adresa celui de-al doilea membru al matricii. Aplicând operatorul de dereferențiere pe acea adresă obținem valoarea stocată acolo (23)

array_pointer_references.c

https://onlinegdb.com/SylW0qlDO

```
valorile celui de-al doilea membru:

*(arr+1)   este: 23
arr[1]     este: 23
*(p+1)     este: 23
```

```
adresele celui de-al doilea
membru:

arr+1        este: 0x7ffd65f990e4
&(arr[1])    este: 0x7ffd65f990e4
p + 1        este: 0x7ffd65f990e4
```

Pointeri și matrici - operatii

```
int arr[4] = {2,23,5,11};
int *p = arr;
```

2	0x7ffd65f990e0	140726314307808
23	0x7ffd65f990e4	140726314307812
5	0x7ffd65f990e8	140726314307816
11	0x7ffd65f990ec	140726314307820

Operațiile cu pointerii (direct, fără dereferențiere) pot fi: adunare și scădere de numere întregi. Compilatorul va presupune că pointerul este parte dintr-o matrice (indiferent dacă a fost astfel declarat sau nu) și se va muta de la o valoare la alta în funcție de numerele adunate sau scăzute.

Operațiile cu pointerii dereferențiați pot fi orice operații care se pot aplica pe valorile conținute

```c
#include <stdio.h>

int main()
{
   int arr[4] = {2,23,5,11};
   int *p = arr;

   printf ("(p + 1) = %p\n", (p+1));
   printf ("*(p + 1) = %d\n\n", *(p+1));

   printf ("(p + 45 - 43) = %p\n", (p + 45 - 43));
   printf ("*(p + 45 - 43) = %d\n\n", *(p + 45 - 43));

   printf ("*(p + 1) + *(p + 3) = %d\n", *(p+1) + *(p + 3));
   printf ("*(p + 1) * *(p + 2) = %d\n", *(p+1) * *(p + 2));
}
```

https://onlinegdb.com/SJUIFPNdN

```
(p + 1) = 0x7ffd65f990e4
*(p + 1) = 23

(p + 45-43) = 0x7ffd65f990e8
*(p + 45 - 43) = 5

*(p + 1) + *(p + 3) = 34
*(p + 1) * *(p + 2) = 115
```

Returnarea unei matrici dintr-o funcție

insertion_copy.c

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

int main()

{
   int data[] = {12,23,5,7,2,9,11,13,9};
   int *sorted = NULL;
   int num = sizeof(data)/sizeof(data[0]);

   printf("Inainte de sort:\n");
   printstr(data,num);
   sorted = insertion(data, num);
   printf("Dupa sort :\n");
   printstr(data,num);
   printf("Sorted :\n");
   printstr(sorted,num);
   return 0;
}
```

https://onlinegdb.com/SygL1OwpQ

Se returnează un pointer către primul element al matricii, fără informație dimensională

```c
int * insertion(int data[], int size)
{
   int *newArr;
   int i,temp,j;
   newArr=(int*)malloc(size*sizeof(int));
   for (i = 0 ; i <= size - 1; i++)
   {
       newArr[i] = data[i];
   }
   for (i = 0 ; i <= size - 1; i++)
   {
     j = i;
    while ( j > 0 && newArr[j] < newArr[j-1])
       {
           temp = newArr[j];
           newArr[j] = newArr[j-1];
           newArr[j-1] = temp;
           j--;
       }
    }
    return newArr;
}
```

```c
void printstr(int data[], int size)
{
    int i;
    for (i = 0; i <= size - 1; i++)
    {
          printf("%d\n", data[i]);
    }
}
```

Returnarea unei matrici dintr-o funcție

insertion_copy.c

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

int main()

{
    int data[] = {12,23,5,7,2,9,11,13,9};
    int *sorted = NULL;
    int num = sizeof(data)/sizeof(data[0]);

    printf("Inainte de sort:\n");
    printstr(data,num);
    sorted = insertion(data, num);
    printf("Dupa sort :\n");
    printstr(data,num);
    printf("Sorted :\n");
    printstr(sorted,num);
    return 0;
}
```

Declarăm un pointer NULL care va primi adresa matricii sortate

Asteptam ceea ce returneaza functia intr-un pointer.

https://onlinegdb.com/SygL1OwpQ

```c
int * insertion(int data[], int size)
{
    int *newArr;
    int i,temp,j;
    newArr=(int*)malloc(size*sizeof(int));
    for (i = 1 ; i <= size - 1; i++)
    {
        newArr[i] = data[i];
    }
    for (i = 1 ; i <= size - 1; i++)
    {
      j = i;
     while ( j > 0 && newArr[j] < newArr[j-1])
        {
            temp = newArr[j];
            newArr[j] = newArr[j-1];
            newArr[j-1] = temp;
            j--;
        }
     }
     return newArr;
}
```

Declarăm un pointer NULL care va primi adresa spațiului de memorie alocat

Acesta NU e pointer! E înmulțire

Returnăm pointerul

```c
void printstr(int data[], int size)
{
    int i;
    for (i = 0; i <= size - 1; i++)
    {
        printf("%d\n", data[i]);
    }
}
```

Indirecție (dereferențiere) multiplă

variabilă	adresă	valoare stocată
a	0x7ffc1d19e2e4	23
p	0x7ffc1d19e2e8	0x7ffc1d19e2e4
pp	0x7ffc1d19e2f0	0x7ffc1d19e2e8
ppp	0x7ffc1d19e2f8	0x7ffc1d19e2f0

indirection.c

```c
#include <stdio.h>

int main()
{
    int a = 23;
    int *p = &a;
    int **pp = &p;
    int ***ppp = &pp;

    printf ("-----Adrese-----\n");
    printf ("a: %p\n", &a);
    printf ("p: %p\n", p);
    printf ("pp: %p\n", pp);
    printf ("ppp: %p\n\n", ppp);

    printf ("-----Valori-----\n");
    printf ("valoare a: %d\n", a);
    printf ("valoare p: %p\n", p);
    printf ("valoare pp: %p\n", pp);
    printf ("valoare ppp: %p\n",
ppp);
    ....
}
```

https://onlinegdb.com/rkMJZsJcN

```
-----Adrese-----
adresa a: 0x7ffc1d19e2e4
adresa p: 0x7ffc1d19e2e8
adresa pp: 0x7ffc1d19e2f0
adresa ppp: 0x7ffc1d19e2f8

-----Valori-----
valoare a: 23
valoare p: 0x7ffc1d19e2e4
valoare pp: 0x7ffc1d19e2e8
valoare ppp: 0x7ffc1d19e2f0
```

Indirecție (dereferențiere) multiplă

variabilă	adresă	valoare stocată
a	0x7ffc1d19e2e4	23
p	0x7ffc1d19e2e8	0x7ffc1d19e2e4
pp	0x7ffc1d19e2f0	0x7ffc1d19e2e8
ppp	0x7ffc1d19e2f8	0x7ffc1d19e2f0

indirection.c

```c
#include <stdio.h>

int main()
{
   int a = 23;
   int *p = &a;
   int **pp = &p;
   int ***ppp = &pp;

....

   printf
("---Dereferentiere---\n");
   printf ("a: %d\n", a);
   printf ("*p: %d\n", *p);
   printf ("**pp: %d\n", **pp);
   printf ("***ppp: %d\n", ***ppp);


}
```

https://onlinegdb.com/rkMJZsJcN

```
-----Dereferentiere-----
a: 23
*p: 23
**pp: 23
***ppp: 23
```

Indirecție (dereferențiere) multiplă - de ce?

```
main()

main(int argc, char * argv[])

main(int argc, char ** argv)

int main()

int main(void)

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

int main(int argc, char ** argv)

int main(int argc, char * argv[], char *envp[])

int main(int argc, char ** argv, char ** envp)
```

argv[] este o matrice de pointeri. Fiecare membru al argv[] este un pointer către un șir de caractere. La transmiterea matricilor de pointeri către funcții este nevoie de dereferențiere multiplă.

Citirea unei variabile din memorie

bytes_to_binary.c

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

 const char * byte to_binary(int x) {
   static char b[9];
   b[0] = '\0';
   int z;
   for (z = 128; z > 0; z >>= 1) {
     strcat(b, ((x & z) == z) ? "1" : "0");
   }
   return b;
 }

int main() {
 int a = 2027433617;
 printf("a este: %i\n", a);
 printf("adresa a: %p, lungime %lu\n", & a, sizeof(a));
 char * p = & a;
 //stack-ul e ordonat invers
 char fb = p[3];
 char sb = p[2];
 char tb = p[1];
 char frb = p[0];
 printf("%s\n", byte to binary(fb));
 printf("%s\n", byte to binary(sb));
 printf("%s\n", byte to binary(tb));
 printf("%s\n", byte_to_binary(frb));
}
```

https://onlinegdb.com/ry5pXgEu4

Cum putem afla reprezentarea binară a unei variabile stocată în memorie?

Declaram a ca fiind un numar intreg (2027433617)

Valoarea respectiva va fi stocata undeva in memorie. Putem afla unde folosind operatorul & (referinta)

Obținem ceea ce se află stocat în 4 adrese de memorie succesive (știm că dimensiunea variabilei a este 4 octeți

Convertim (pentru afișare) ceea ce am găsit în cele 4 adrese în baza 2.

Ne așteptăm ca prin concatenarea celor 4 “fragmente” de cod binar să obținem corespondentul în baza 2 al numărului declarat în variabila a (2027433617).

Paranteză...

bytes_to_binary.c

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

```c
const char * byte to_binary(int x) {
  static char b[9];
  b[0] = '\0';
  int z;
  for (z = 128; z > 0; z >>= 1) {
    strcat(b, ((x & z) == z) ? "1" : "0");
  }
  return b;
}
```

```c
int main() {
 int a = 2027433617;
 printf("a este: %i\n", a);
 printf("adresa a: %p, lungime %lu\n", & a, sizeof(a));
 char * p = & a;
 //stack-ul e ordonat invers
 char fb = p[3];
 char sb = p[2];
 char tb = p[1];
 char frb = p[0];
 printf("%s\n", byte to binary(fb));
 printf("%s\n", byte to binary(sb));
 printf("%s\n", byte to binary(tb));
 printf("%s\n", byte_to_binary(frb));
}
```

https://onlinegdb.com/ry5pXgEu4

Funcție care returnează conversia binară a unui char (unui singur octet).

E interesantă declarația for, care folosește operatorul right shift în loc de împărțire. Este echivalenta cu:

```c
for (z = 128; z > 0; z = z/2)
```

De asemenea, instructiunea:

```c
strcat(b, ((x & z) == z) ? "1" : "0");
```

(x & z) == z) testeaza daca versiunea binară a numărului X conține un 1 pe poziția corespunzătoare 1-ului din z

x este: 46; z este: 32; **x & z este: 32**

x binar este: 00101110
z binar este: 00100000
x & z binar este: 00100000

toate z-urile fiind putere a lui 2 au un singur 1

Citirea unei variabile din memorie

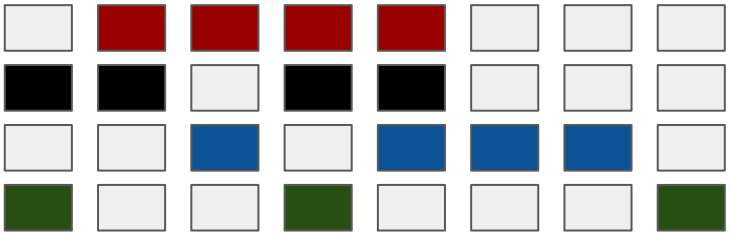
bytes_to_binary.c

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

 const char * byte to_binary(int x) {
   static char b[9];
   b[0] = '\0';
   int z;
   for (z = 128; z > 0; z >>= 1) {
     strcat(b, ((x & z) == z) ? "1" : "0");
   }
   return b;
 }

int main() {
 int a = 2027433617;
 printf("a este: %i\n", a);
 printf("adresa a: %p, lungime %lu\n", & a,
sizeof(a));
 char * p = & a;
 //stack-ul e ordonat invers
 char fb = p[3];
 char sb = p[2];
 char tb = p[1];
 char frb = p[0];
 printf("%s\n", byte to binary(fb));
 printf("%s\n", byte to binary(sb));
 printf("%s\n", byte to binary(tb));
 printf("%s\n", byte_to_binary(frb));
}
```

https://onlinegdb.com/ry5pXgEu4

+2027433617

01111000110110000010111010010001



```
a este: 2027433617
adresa a: 0x7ffc8509eea4, lungime
4
octet 1: 01111000
octet 2: 11011000
octet 3: 00101110
octet 4: 10010001
```

01111000110110000010111010010001

Pointeri definiți “greșit”

```
char * p = &a;
```

Pentru a obține ceea ce se află într-o anumită adresă de memorie am definit un tip special de variabila care se numește pointer.

Un pointer este un tip special de variabilă care stochează adresa unei locații de memorie și permite obținerea datelor stocate în acea locație.

Adresa unei locații de memorie este un număr întreg, reprezentat de cele mai multe ori în format hexazecimal. Un număr întreg poate fi stocat în tipul de date corespunzător numerelor întregi (ex: int, long) în funcție de arhitectura.

Pointer-ul face ceva în plus: permite accesul direct la datele stocare în acea adresă de memorie.

Pentru a avea acces la date, pointerul trebuie sa ‘’știe” câte adrese de memorie trebuie sa citească și să “lipească” pentru a returna valoarea corectă.

De aceea, pointeriii se definesc cu tipul variabilei corespunzătoare (aici char)

Dar… `int a = 2027433617;`

`a este int, nu ar fi trebuit să definim:`

```
int * p = &a; ???
```

Ba da, în mod normal așa vom face. În acest caz, pentru a exemplifica citirea “octet cu octet” a fost nevoie sa “păcălim” compilatorul si a trebuit să declarăm un pointer către un sfert din a.

De alftel, la compilarea programului avem un warning:

```
warning: initialization of 'char *' from incompatible pointer type 'int *'
```

Pointeri ca matrici

```
char * p = & a;
char fb = p[3];
char sb = p[2];
char tb = p[1];
char frb = p[0];
```

O caracteristică a pointerilor este că pot fi apelați cu indice, ca și cum ar fi o matrice, indiferent de definirea unei matrici sau nu. Rezultatul este ceea ce este stocat în următoarea adresă de memorie.

Dacă pointerul ar fi fost de tip int, rezultatul ar fi fost următoarele 4 adrese de memorie, unite pentru a deveni numărul întreg corespunzător.

Variabila noastră a avea doar 4 octeți. Pointerul nostru, fiind de tip char va accesa un singur octet. Prin apelarea succesiva a pointerului cu index-ul între 0 și 3 obținem toți octeții în care este stocat numărul a.

Putem solicita conținutul adreselor de memorie dincolo de variabila a, pointerul le returnează fără probleme. Pointerul este de tip char și nu are nici o legătură cu faptul că în primele 4 adrese este stocată o variabilă, el returnează ceea ce îi solicităm. Evident, ceea ce se găsește dincolo de al 4-lea octet nu are sens pentru noi.

Ce se întâmplă dacă mergem mai departe de al 4-lea octet?

octet 1: 10010001
octet 2: 00101110
octet 3: 11011000
octet 4: 01111000
octet 5: 00111100
octet 6: 01001110
octet 7: 00011001
octet 8: 00001101
octet 9: 11111101
octet 10: 01111111
octet 11: 00000000

Pointeri generici și utilizarea lor

Un pointer stochează o adresă de memorie, indiferent de tipul acestuia.

Tipul pointerului este necesar doar pentru operația de derefderențiere. Trebuie să știm câți octeți “citim” din memorie începând cu adresa stocată în pointer.

Dacă NU avem nevoie de dereferențiere, putem renunța la tip. Anumite funcții NU au nevoie de operatia de dereferențiere în interiorul lor, astfel, acestea pot opera doar cu adresele stocate. Acestea se numesc funcții generice pentru că pot primi orice fel de pointer.

Evident, după aplicarea funcției, rezultatul acesteia va trebui utilizat în alte operații și va avea nevoie de dereferențiere. Aceasta se va face prin convertirea pointerului către tipul cerut.

```
void *malloc(size_t size)
```

Funcția malloc alocă o anumită cantitate de memorie (în heap). Malloc are nevoie de un singur parametru, numărul de octeți pe care dorim să ii obținem.

Malloc va returna un pointer către baza zonei de memorie alocată. În această zonă de memorie vom dori să scriem variabile, acestea vor avea un tip. Malloc nu trebuie să știe ce vom scrie în zona respectivă dar trebuie să ne ofere un pointer către baza acelei zone. De aceea, malloc returnează un pointer generic (doar adresa, fără tip), pe care va trebui să îl convertim la tipul de pointer care ne trebuie.

```
newArr=(int*)malloc(size*sizeof(int));
```

size_t (stdlib.h)

size_t este un tip de date definit în stdlib.h care este returnat de operatorul sizeof

Acesta este un tip de date asemănător cu unsigned int, definit în funcție de arhitectura sistemului de operare.

size_t este garantat ca fiind suficient de mare pentru a conține dimensiunea oricărei variabile definite.

size_t nu este niciodată negativ.

size_t este utilizat în multe situații de funcțiile frecvent utilizate în c.

Pointeri către funcții

În momentul încărcării unui program în memorie, fiecare funcție (procedură, subrutină) este încărcată într-un spațiu de memorie care începe la o anumită adresă. Apoi, în corpul programului (și a altor funcții) se înlocuiește apelul funcției respective cu o instrucțiune de salt la adresa respectivă.

Adresa funcției poate fi obținută în program (ca și adresa variabilelor) cu ajutorul unui pointer către funcție.

Definirea normală a funcției display care primește un int

Funcția “executor” primește argumentul “ptr” care este un pointer către funcția display

se atribuie unui pointer o referinta catre functia display

se lansează funcția executor cu argumentul ptr care este un pointer către funcția “display”

pointer_to_functions.c

```c
#include<stdio.h>

void display(int n)
{
   printf("Eu sunt functia display si am primit %d\n", n);
}

void executor(void (*ptr)())
{
   int mi = 13;
   (*ptr) (mi); // apelul funcției din spatele pointerului ptr
}

int main()
{

   void (*ptr)() = &display;

   executor(ptr);
   return 0;
}
```

https://onlinegdb.com/Skx-Dw6Iu

Pointeri către funcții - qsort (stdlib.h)

void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))

base: pointer către baza matricii care trebuie sortată

size: dimensiunea fiecărui element din matrice (în octeți)

nitems: numărul de elemente din matricea care trebuie sortată

compar: pointer către funcția care compară două elemente

Funcția compar trebuie să aibă două argumente de tip const void * (pointer generic).

qsort funcționează direct asupra matricii trimise; după rularea funcției aceasta va fi modificată (sortată), ordinea inițială nu se va păstra nicăieri.

Funcția qsort este de tipul “void” - nu returnează nimic

qsort este o funcție impură

qsort - funcția de comparație

Funcția de comparație trebuie să primească doi pointeri generici către variabilele de comparat

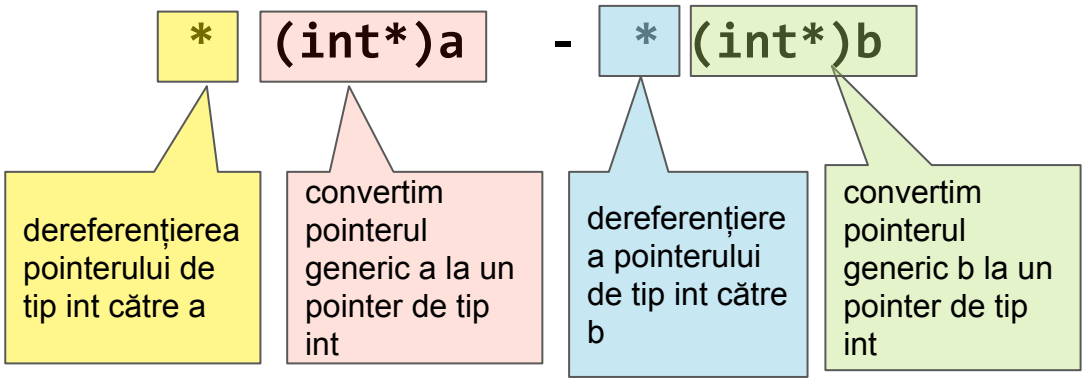
Funcția de comparație trebuie să returneze:

- o valoare negativă dacă primul argument este mai mic decât cel de-al doilea,
- o valoare pozitivă dacă primul argument este mai mare decât cel de-al doilea

```
int cmpfunc (const void * a, const void * b) {
    return ( *(int*)a - *(int*)b );
}
```

Funcția cmpfunc trebuie să aibă pointeri generici ca argumente pentru ca funcția qsort să poată sorta orice tip de date. qsort este astfel o **funcție generică**.

cmpsort însă nu poate fi funcție generică - știm că vom avea de sortat numere întregi și știm că pentru a calcula diferența dintre ele trebuie să dereferențiem pointerii trimiși către cmpsort. Știm și că pointerii generici NU pot fi dereferențiați, de aceea este nevoie să convertim pointerii generici în pointeri de tip int.



ptr_deref.c

```
int main () {
    int a = 13;
    int *ptr = &a;
    printf("%d", *(int*)ptr
);
}
```

https://onlinegdb.com/qkNFZXLrA

Pointeri către funcții - qsort (stdlib.h)

void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))

base: pointer către baza matricii care trebuie sortată

nitems: numărul de elemente din matricea care trebuie sortată

size: dimensiunea fiecărui element din matrice (în octeți)

compar: pointer către funcția care compară două elemente

Apelul funcției qsort

qsort.c

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


int cmpfunc (const void * a, const void * b) {
  return ( *(int*)a - *(int*)b );
}

int main () {
  int n;
  int values[] = { 88, 56, 100, 2, 25 };

  printf("Inainte de sortare: \n");
  for( n = 0 ; n < 5; n++ ) {
     printf("%d ", values[n]);
  }

  qsort(values, 5, sizeof(int), cmpfunc);

  printf("\nDupa sortare: \n");
  for( n = 0 ; n < 5; n++ ) {
     printf("%d ", values[n]);
  }

  return(0);
}
```

https://onlinegdb.com/rkcZowalu

Pointeri către funcții - bsearch (stdlib.h)

```
void *bsearch(
    const void *key,
    const void *base,
    size_t nitems,
    size_t size,
    int (*compar)(const void *, const void *)
)
```

- const void *key, — Pointer generic către variabila care conține elementul care trebuie căutat
- const void *base, — Pointer generic către baza matricii
- size_t nitems, — Numărul de elemente din matrice
- size_t size, — Dimensiunea unui element al matricii
- int (*compar)(const void *, const void *) — Funcția de comparație

Numărul de elemente din array

Apelul funcției bsearch

bsearch.c

```c
int main () {
   int numere[] = { 5, 20, 29, 32, 63, 68, 97, 121, 145, 212 };
   int key = 32;
   int vsize = sizeof(numere)/sizeof(numere[0]);

   int *f = (int*) bsearch(&key, numere, vsize, sizeof(numere[0]), cmpfunc);

   if( f != NULL ) {
         printf("Am gasit = %d\n", *f);
   } else {
         printf("Nu am gasit %d \n", key);
   }
   return(0);
}
```

https://onlinegdb.com/HJa0gzALu

Funcțiile pot returna pointeri generici

Pointer generic

```
void *bsearch(const void *key, const void *base, size_t nitems,
size_t size, int (*compar)(const void *, const void *))
```

Funcția returnează un pointer generic pentru că este proiectată să returneze un pointer către elementul găsit. Astfel, pointerul returnat trebuie să fie convertit în tipul corespunzător elementelor care trebuie sortate.

```
int *f = (int*) bsearch(&key, numere, vsize, sizeof(numere[0]), cmpfunc);
```

Dacă nu se găsește elementul căutat, funcția va returna un pointer NULL. De aceea, trebuie să verificăm pointerul returnat.

```
if( f != NULL ) {
   printf("Am gasit = %d\n", *f);
} else {
   printf("Nu am gasit %d \n", key);
}
```

Șiruri de caractere

șir n. **1.** serie de ființe sau lucruri așezate în linie: *un șir de case, un șir de soldați;* **2.** serie, succesiune: *un șir de nenorociri;* **3.** rang, rând: *îl primesc în șirurile lor;* **4.** ordine, legătură logică: *vorbire fără șir;* **5.** înșirare: *șir de mărgăritare.* [Ung. SER].

- sursa: Șăineanu, ed. VI (1929)

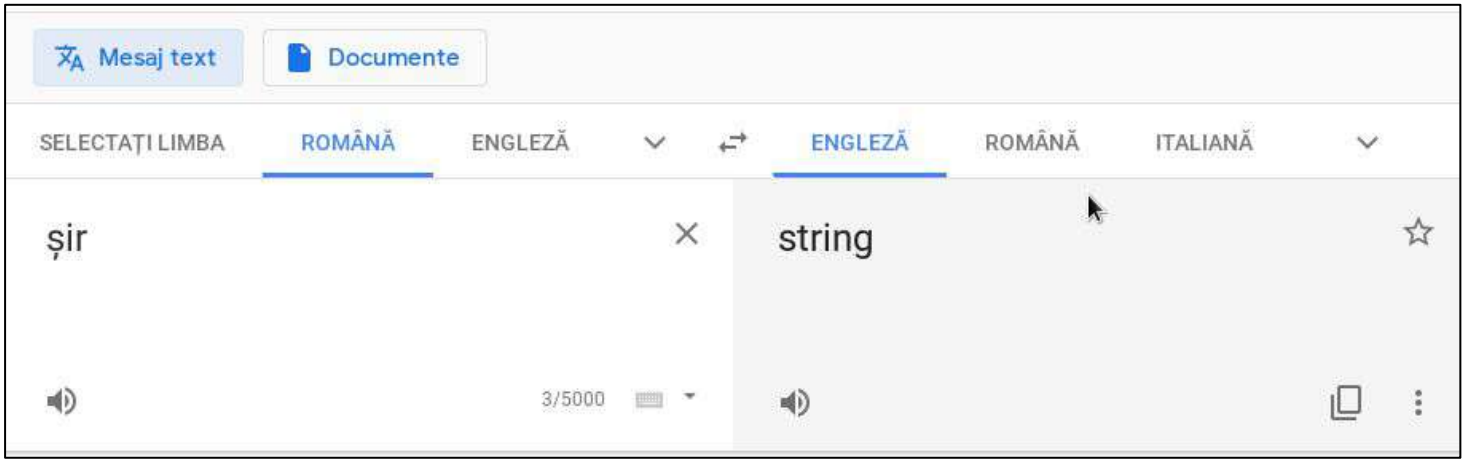
Un șir reprezintă o colecție unidimensională continuă de date identice

Șiruri in C

- Un șir în C este o colecție de variabile care sunt stocate în locații de memorie succesive.
- Un șir poate conține un singur tip de variabile
- Un șir poate conține orice tip de variabilă, atâta timp cât toți membrii săi sunt la fel.
- Dimensiunea șirului (numărul de variabile pe care le conține) trebuie să fie cunoscută la momentul definirii acestuia.
- Elementele din șir se pot accesa cu ajutorul indexului. sir[0] este primul element, sir[1] al doilea etc.
- Adresa șirului este adresa primului element al acestuia (elementul cu index 0).
- Șirurile au posibilitatea de a fi traversate prin folosirea indexului.

```
for ( i = 0; i < N; i++ ) {
  printf("Valoarea lui %i: %d\n",i, val[i]);
}
```

Șir / string



în limba engleză sensul cuvântului string care este cea mai directă traducere a cuvântului șir are un sens mult mai strict (în special în context informatic) - în marea majoritate a cazurilor înseamnă șir de caractere

Șiruri de caractere în C

Un șir de caractere în C este un șir de date de tip char care conține un număr de caractere egal cu numărul de caractere din textul atribuit la care se adaugă un caracter suplimentar numit caracter nul (null character, '\0'). Terminatorul null corespunde cu codul ASCII 0 (nu codul caracterului 0 care este 48).

c	a	r	a	c	a	t	i	t	a	\0
1	2	3	4	5	6	7	8	9	10	11

https://onlinegdb.com/SyWBvkGw_

```c
#include <stdio.h>

int main()
{
  char stra[] = {'c', 'a', 'r', 'a', 'c', 'a', 't', 'i', 't', 'a'};
  char strb[] = "caracatita";

  printf ("stra: %ld\n", sizeof(stra)/sizeof(stra[0]));
  printf ("strb: %ld\n", sizeof(strb)/sizeof(strb[0]));
  return 0;
}
```

sir de char fără terminație

terminația este adăugată automat la declarația șirului de caractere în acest mod

siruri1.c

```
a: 10
b: 11
```

Dimensiunea unui șir: strlen

int strlen(const char *str);

Terminatorul șirului nu este printabil dar ocupă o poziție din șir. Astfel, un șir de caractere definit ca având lungimea 50 va putea conține în realitate 49 de caractere “utile”.

https://onlinegdb.com/ds8jPMDo8

```c
#include <stdio.h>
#include <string.h>

int main()
{
  char strb[] = "caracatita";
  int i;

  printf ("sizeof returneaza: %ld\n", sizeof(strb));
  printf ("strlen returneaza: %ld\n", strlen(strb))

  for (i = 0; i < sizeof(strb); i++) {
    printf ("cr: %c numar: %d\n", strb[i], strb[i]);
  }

  return 0;
}
```

siruri2.c

```
sizeof returneaza:
11
strlen returneaza:
10
cr: c - numar: 99
cr: a - numar: 97
cr: r - numar: 114
cr: a - numar: 97
cr: c - numar: 99
cr: a - numar: 97
cr: t - numar: 116
cr: i - numar: 105
cr: t - numar: 116
cr: a - numar: 97
cr:  - numar: 0
```

sizeof este un operator care se aplica pe orice șir/matrice și nu “știe” că este vorba de un șir de caractere; de aceea va returna dimensiunea sa reala, incluzând ultimul caracter (terminatorul nul).

Biblioteca string.h conține funcții speciale de operare cu șiruri de caractere care știu să evite terminatorul null în operații (**strlen** returneaza dimensiunea corectă)

Dimensiunea unui șir: strlen

int strlen(const char *str);

siruri3.c

```c
#include <stdio.h>
#include <string.h>

int main()
{

  char stra[] = {'c', 'a', 'r','a','c','a','t','i','t','a'};
  char strb[] = "caracatita";
  char strc[] = {'c', 'a', 'r','a','c','a','t','i','t','a','\0'};

  printf ("stra: %ld\n", sizeof(stra)/sizeof(stra[0]));
  printf ("strb: %ld\n", sizeof(strb)/sizeof(strb[0]));
  printf ("strc: %ld\n", sizeof(strc)/sizeof(strc[0]));
  printf ("strlen stra: %ld\n", strlen(stra));
  printf ("strlen strb: %ld\n", strlen(strb));
  printf ("strlen strc: %ld\n", strlen(strc));

  return 0;
}
```

https://onlinegdb.com/r1U1_yfvO

```
stra: 10
strb: 11
strc: 11
strlen stra: 20
strlen strb: 10
strlen strc: 10
```

strlen stra: 20 arată o dimensiune nepotrivită. În lipsa terminatorului nul, funcțiile care manevrează șiruri de caractere dau rezultate neașteptate și de multe ori erori, în acest caz rezultatul depinde mult de compilator.

Probabil că stra și strb sunt una în continuarea celeilalte în stack iar strlen se oprește la terminatorul nul lui strb.

#include <string.h>

memchr 201	**void *memchr(const void *str, int c, size_t n)** Caută prima apariție a caracterului c (unsigned char) in primii n octeți din șirul către care trimite pointerul str
memcmp 2661	**int memcmp(const void *str1, const void *str2, size_t n)** Compară primii n octeți ai șirurilor către care trimit pointerii str1 și str2
memcpy 23382	**void *memcpy(void *dest, const void *src, size_t n)** Copiază direct n octeți din sursă în destinație
memmove 979	**void *memmove(void *dest, const void *src, size_t n)** Copiază cu ajutorul unui buffer intermediar n octeți din sursă în destinație
memset 17258	**void *memset(void *str, int c, size_t n)** Copiază caracterul c (unsigned char) în primele n caractere din șirul către care trimite pointerul str.
strcat 578	**char *strcat(char *dest, const char *src)** Adaugă șirul de caractere către care trimite pointerul *src la finalul șirului de caractere către care trimite pointerul *dest

memchr, memcmp, memcpy, memmove și **memset** sunt funcții care nu operează neapărat cu șiruri de caractere ci cu șiruri de octeți. Aceasta se vede prin faptul că argumentele de intrare sunt **pointeri generici**, nu **pointeri de tip char**, ca la restul funcțiilor din string.h.

Aceste funcții sunt prezente în string.h oarecum din întâmplare (au apărut mai târziu in C și string.h era o bibliotecă folosită de toată lumea)

201

Apariții ale funcției în sursa nucleului Linux (inclusiv comentarii)

#include <string.h>

strncat 102	**char *strncat(char *dest, const char *src, size_t n)** Adaugă n caractere din șirul de caractere către care trimite pointerul *src la finalul șirului de caractere către care trimite pointerul *dest
strchr 795	**char *strchr(const char *str, int c)** Caută prima apariție a caracterului c (unsigned char) în șirul de caractere către care trimite pointerul *str*.
strcmp 5413	**int strcmp(const char *str1, const char *str2)** Compară șirul de caractere către care trimite pointerul str1 cu cel către care trimite pointerul str2.
strncmp 2943	**int strncmp(const char *str1, const char *str2, size_t n)** Compară primii n octeți din șirul de caractere către care trimite pointerul str1 cu primii n octeți din cel către care trimite pointerul str2.
strcoll	**int strcoll(const char *str1, const char *str2)** Compară șirurile de caractere către care trimit pointerii str1 și str2 ținând cont de setările locale (LC_COLLATE)
strcpy 2712	**char *strcpy(char *dest, const char *src)** Copiază șirul de caractere către care trimite pointerul src în șirul de caractere către care trimite pointerul dest.

#include <string.h>

strncpy 1263	**char *strncpy(char *dest, const char *src, size_t n)** Copiază n caractere sin șirul de caractere către care trimite pointerul src în șirul de caractere către care trimite pointerul dest, fără a adăuga terminatorul NULL.
strcspn 48	**size_t strcspn(const char *str1, const char *str2)** Calculează lungimea segmentului inițial al șirului de caractere str1 care conține doar caractere care nu sunt prezente în str2.
strerror 517	**char *strerror(int errnum)** https://onlinegdb.com/vf-ts_9BR Returnează un pointer către un mesaj de eroare corespunzător variabilei errnum stocat într-o matrice internă.
strlen 4491	**size_t strlen(const char *str)** Calculează lungimea șirului de caractere către care trimite pointerul str fără să includă terminația nulă
strpbrk 31	**char *strpbrk(const char *str1, const char *str2)** Caută primul caracter în șirul de caractere către care trimite pointerul str1 care corespunde cu orice caracter specificat în șirul de caractere către care trimite str2
strrchr 145	**char *strrchr(const char *str, int c)** Caută ultima apariție a caracterului c în șirul către care trimite str

#include <string.h>

strspn 36	**size_t strspn(const char *str1, const char *str2)** Calculează lungimea segmentului inițial al șirului de caractere către care trimite str1 care constă doar din caractere care se găsesc în șirul de caractere către care trimite str2.
strstr 543	**char *strstr(const char *haystack, const char *needle)** Găsește prima apariție a întregului șir de caractere către care trimite pointerul needle (fără terminatorul nul) care apare în șirul de caractere către care trimite pointerul *haystack*.
strtok 62	**char *strtok(char *str, const char *delim)** Sparge șirul de caractere către care trimite pointerul str într-o serie de fragmente pe baza șirului de caractere către care trimite pointerul delim
strxfrm	**size_t strxfrm(char *dest, const char *src, size_t n)** Transformă primele n caractere din șirul de caractere către care trimite pointerul src ținând cont de setarea locale curentă și le scrie în șirul de caractere către care trimite pointerul dest.

Atribuirea șirurilor de caractere

Atribuirea șirurilor de caractere se poate face fie la declararea acestora, fie după declarare.

După declarare, atribuirea se poate face doar caracter cu caracter sau prin utilizarea funcțiilor de tipul strcpy. Pentru că un șir de caractere este o matrice și matricile în C sunt ne-atribuibile.

https://onlinegdb.com/SJB7_kzvd

```c
#include <stdio.h>
#include <string.h>

int main()
{

  char stra[] = {'c', 'a', 'r','a','c','a','t','i','t','a'};
  stra = "nautilus";
  printf ("stra este: %s\n", stra);

  return 0;
}
```

siruri4.c

```
string_asignment.c: In function 'main':
string_asignment.c:9:9: error: assignment to expression with array type
     stra = "nautilus";
     ^
```

Șiruri de caractere și pointeri

https://onlinegdb.com/BJVcu1zv_

```c
#include <stdio.h>

int main(void) {

 char *sirPtr = "Cine a pus ulei pe clanta";

 printf("%s\n", sirPtr);

 sirPtr++;
 printf("%s\n", sirPtr);
 sirPtr++;
 printf("%s\n", sirPtr);
 sirPtr++;
 printf("%s\n", sirPtr);
 sirPtr--;
 printf("%s\n", sirPtr);
 sirPtr--;
 printf("%s\n", sirPtr);
 sirPtr--;
 printf("%s\n", sirPtr);

 return 0;
}
```

"%s" nu are nevoie de dereferențiere pentru că se așteaptă la un pointer

pointer_string1.c

```
Cine a pus ulei pe clanta
ine a pus ulei pe clanta
ne a pus ulei pe clanta
e a pus ulei pe clanta
ne a pus ulei pe clanta
ine a pus ulei pe clanta
Cine a pus ulei pe clanta
```

https://onlinegdb.com/ryjkF1GwO

```c
#include <stdio.h>

int main(void) {
 char *sirPtr = "Cine a pus ulei pe clanta";

 printf("%s\n", sirPtr);

 printf("%c\n", *sirPtr);
 sirPtr++;
 printf("%c\n", *sirPtr);
 sirPtr++;
 printf("%c\n", *sirPtr);
 sirPtr++;
 printf("%c\n", *sirPtr);
 sirPtr--;
 printf("%c\n", *sirPtr);
 sirPtr--;
 printf("%c\n", *sirPtr);
 sirPtr--;
 printf("%c\n", *sirPtr);

 return 0;
}
```

"%c" are nevoie de dereferențiere pentru că așteaptă un char, nu un pointer.

pointer_string2.c

```
Cine a pus ulei pe clanta
C
i
n
e
n
i
C
```

Șiruri de caractere și pointeri

pointer_string3.c

```c
#include <stdio.h>

int main(void) {

 char *sirPtr = "Cine a pus ulei pe clanta";

 printf("%s\n", sirPtr);
 printf("%s\n", sirPtr+1);
 printf("%s\n", sirPtr+7);



 printf("%c\n", *sirPtr);
 printf("%c\n", *(sirPtr+1));
 printf("%c\n", *(sirPtr+7));
 return 0;
}
```

https://onlinegdb.com/H1t8K1zvu

```
Cine a pus ulei pe clanta
ine a pus ulei pe clanta
pus ulei pe clanta
C
i
p
```

%s (string), sirPtr (pointer către primul caracter din șir - de fapt sirPtr + 0). Se va imprima până la întâlnirea terminatorului NULL.

%s (string), sirPtr + 1 (pointer către al doiea caracter din șir). Se va imprima până la întâlnirea terminatorului NULL.

%s (string), sirPtr + 7 (pointer către al optulea caracter din șir). Se va imprima până la întâlnirea terminatorului NULL.

%c (caracter), *sirPtr (dereferentierea pointerului către primul caracter din șir). Se va imprima caracterul către care trimite pointerul.

%c (caracter), *(sirPtr+1) (dereferentierea pointerului către cel de-al doilea caracter din șir). Se va imprima caracterul către care trimite pointerul.

%c (caracter), *(sirPtr+1) (dereferentierea pointerului către cel de-al doilea caracter din șir). Se va imprima caracterul către care trimite pointerul.

Modificarea șirurilor de caractere

siruri5.c

```c
#include <stdio.h>
#include <string.h>

int main()
{

  char stra[] = {'c', 'a', 'r','a','c','a','t','i','t','a',
'\0'};
  stra[0] = 'n';
  stra[1] = 'a';
  stra[2] = 'u';
  stra[3] = 't';
  stra[4] = 'i';
  stra[5] = 'l';
  stra[6] = 'u';
  stra[7] = 's';

  printf ("stra este: %s\n", stra);

  return 0;
}
```

https://onlinegdb.com/SJ7Ey3D94

c	n
a	a
r	u
a	t
c	i
a	l
t	u
i	s
t	t
a	a
0	0

```
stra este: nautilusta
```

stra conține inițial cuvântul “caracatiță” (10 caractere). Cuvântul nautilus conține doar 8 caractere - celelalte două poziții rămân ocupate de ultimele două caractere din cuvântul caracatita.

Erori

siruri6.c

```c
#include <stdio.h>
#include <string.h>

int main()
{
  char stra[] = {'c', 'a', 'r','a','c','a','t','i','t','a'};
  stra[0] = "n";
  stra[1] = "a";
  stra[2] = 'u'; stra[3] = 't'; stra[4] = 'i';
  stra[5] = 'l';stra[6] = 'u';stra[7] = 's';

  printf ("stra[0] este: %d\n", stra[0]);
  printf ("stra[1] este: %d\n", stra[1]);
  printf ("stra[2] este: %d\n", stra[2]);
  printf ("stra este: %s\n", stra);
  return 0;
}
```

Single quotes!!

https://onlinegdb.com/SkMmG3wcE

```
string_assignment2.c: In function 'main':
string_assignment2.c:8:12: warning: assignment to 'char' from 'char *' makes integer from
pointer without a cast [-Wint-conversion]
     stra[0] = "a";
               ^
string_assignment2.c:9:12: warning: assignment to 'char' from 'char *' makes integer from
pointer without a cast [-Wint-conversion]
     stra[1] = "n";
               ^
```

```
stra[0] este:
stra[1] este:
stra[2] este: u
stra este: utilusta
```

`stra[0] = "n";` atribuie “n” care nu reprezintă un caracter ci un șir de caractere. Adică un pointer constant către un șir format din caracterul a și terminatorul null.

char *strcpy(char *dest, const char *src)

Copiază șirul de caractere către care trimite pointerul src în șirul de caractere către care trimite pointerul dest.

siruri7.c

```c
#include <stdio.h>
#include <string.h>

int main () {
  char str1[20];
  char str2[20];

  strcpy(str1, "caracatita");
  strcpy(str2, "nautilus");

  printf ("str1 este: %s\n", str1);
  printf ("str2 este: %s\n", str2);

  return 0;
}
```

https://onlinegdb.com/SJKwjyMPO

```
str1 este: caracatita
str2 este: nautilus
```

https://onlinegdb.com/B1uWBhw9V

strcpy este frecvent folosită ca funcție de atribuire pentru șiruri.

La declararea unui șir de caractere:

```c
char str1[20];
```

compilatorul va inițializa un șir de 20 de char și le va umple cu caracterul terminator nul.

```c
strcpy(str1, "caracatita");
```

va copia șirul de caractere “caracatita” în primele 11 (10 + terminatorul null) în str1.

char *strcpy(char *dest, const char *src)

https://onlinegdb.com/BkguIhP9E

```c
#include <stdio.h>
#include <string.h>

int main()
{

  char stra[] = "caracatita";
  char strb[] = "nautilus";

  printf ("strlen stra este: %ld\n", strlen(stra));
  printf ("sizeof stra este: %ld\n", sizeof(stra));

  strcpy(stra, strb);
  printf ("stra este: %s\n", stra);
  printf ("strlen stra este: %ld\n", strlen(stra));
  printf ("sizeof stra este: %ld\n", sizeof(stra));
  printf ("stra[8] este: %c\n", stra[8]);
  printf ("stra[9] este: %c\n", stra[9]);
  return 0;
}
```

siruri8.c

```
strlen stra este: 10
sizeof stra este: 11
stra este: nautilus
strlen stra este: 8
sizeof stra este: 11
stra[8] este:
stra[9] este: a
```

null

strcpy va copia inclusiv terminatorul nul astfel încât, deși dimensiunea matricii corespunzătoare lui **stra** va rămâne 11, strlen (care numără până la primul terminator null) va returna 8 iar sprintf, care de asemenea se oprește la primul terminator va afișa doar “nautilus”. După terminarea șirului sursă (inclusiv null) celelalte caractere (dacă șirul destinație este mai lung) rămân la fel.

char *strncpy(char *dest, const char *src, size_t n)

siruri9.c

```c
#include <stdio.h>
#include <string.h>

int main()
{

  char stra[] = "caracatita";
  char strb[] = "nautilus";

  printf ("strlen stra este: %ld\n", strlen(stra));
  printf ("sizeof stra este: %ld\n", sizeof(stra));

  strncpy(stra, strb,8);
  printf ("stra este: %s\n", stra);
  printf ("strlen stra este: %ld\n", strlen(stra));
  printf ("sizeof stra este: %ld\n", sizeof(stra));
  printf ("stra[8] este: %c\n", stra[8]);
  printf ("stra[9] este: %c\n", stra[9]);
  return 0;
}
```

https://onlinegdb.com/ryp383vcN

```
strlen stra este: 10
sizeof stra este: 11
stra este: nautilusta
strlen stra este: 8
sizeof stra este: 11
stra[8] este: t
stra[9] este: a
```

strncpy copiază doar numărul de caractere corespunzător parametrului de intrare n. NU adaugă terminator nul. De aceea, ultimele două caractere rămân parte din șir.

probleme în utilizarea strcpy

siruri10.c

```c
#include <stdio.h>
#include <string.h>

int main()
{

  char stra[] = "caracatita";
  char strb[] = "nautilus nu este octopod";
  char strc[] = "ambele sunt cefalopode";

  printf ("strlen stra este: %ld\n", strlen(stra));
  printf ("sizeof stra este: %ld\n", sizeof(stra));

  strcpy(stra, strb);
  printf ("stra este: %s\n", stra);
  printf ("strc este: %s\n", strc);
  printf ("strlen stra este: %ld\n", strlen(stra));
  printf ("sizeof stra este: %ld\n", sizeof(stra));

  return 0;
}
```

https://onlinegdb.com/rk45PnDqN

Utilizarea funcției strcpy trebuie făcută cu mare atenție pentru a nu solicita copierea unui șir mai mare într-un șir mai mic.

Consecința se numește “buffer overrun”

```
strlen stra este: 10
sizeof stra este: 11
stra este: nautilus nu este octopod
strc este:  este octopod
strlen stra este: 24
sizeof stra este: 11
```

Faptul că strb este mai lung decât stra face ca o parte din conținutul său să intre peste următoarea variabilă din stack (strc) și să o suprascrie complet. Comportamentul depinde de compilator si executie.

probleme în utilizarea strncpy

siruri11.c

```c
#include <stdio.h>
#include <string.h>

int main()
{

  char stra[] = "caracatita";
  char strb[] = "nautilus nu este octopod";
  char strc[] = "ambele sunt cefalopode";

  printf ("strlen stra este: %ld\n", strlen(stra));
  printf ("sizeof stra este: %ld\n", sizeof(stra));

  strncpy(stra, strb, 16);
  printf ("stra este: %s\n", stra);
  printf ("strc este: %s\n", strc);
  printf ("strlen stra este: %ld\n", strlen(stra));
  printf ("sizeof stra este: %ld\n", sizeof(stra));

  return 0;
}
```

https://onlinegdb.com/SktN_hP9E

strncpy nu adaugă de la sine un terminator nul.

Utilizarea neatentă conduce din nou la buffer overrun.

```
strlen stra este: 10
sizeof stra este: 11
stra este: nautilus nu estee sunt
cefalopode
strc este:  estee sunt cefalopode
strlen stra este: 33
sizeof stra este: 11
```

16 caractere este mai puțin decât lungimea lui strb (24) dar mai mult decât lungimea lui stra (10). Consecință: nu mai există nici un terminator nul. Primul disponibil este abia la finalul șirului de caractere strc care se găsește în stack după strb. Ca atare, citirea șirului de caractere stra dă rezultate false (un fel de concatenare între strb și ceea ce a rămas din strc).

Citirea șirurilor de la consolă

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

este o funcție generală care permite citirea de la consolă a unor date și atribuirea acestora unei variabile.

scanf1.c

```c
#include <stdio.h>

int main()
{
   int s;
   float b;
   char c;
   printf("Introduceti un numar intreg:");
   scanf ("%d", &s);
   printf ("Ati introdus numarul %d\n",s);
   printf("Introduceti un numar rational:");
   scanf ("%f", &b);
   printf ("Ati introdus numarul %f\n",b);
   printf("Introduceti un caracter:");
   scanf (" %c", &c);
   printf ("Ati introdus caracterul %c\n",c);
   return 0;
}
```

https://onlinegdb.com/ry_y_P92X

```
Introduceti un numar intreg:12
Ati introdus numarul 12
Introduceti un numar rational:32.45
Ati introdus numarul 32.450001
Introduceti un caracter:d
Ati introdus caracterul d
```

Citirea șirurilor de la consolă

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

Până acum am utilizat scanf pentru citirea numerelor. Variabilele în care așteptam să se regăsească valorile citite trebuiau transmise către scanf ca referințe pentru a fi disponibile și după încheierea execuției funcției.

https://onlinegdb.com/rk8WU6xwO

scanf2.c

```c
#include <stdio.h>
#include <string.h>

int main()
{
  char str[100];
  printf("introduceti o expresie:");
  scanf("%s", str);
  printf("introdus: %s",str);
  return 0;
}
```

```
introduceti o expresie:cine a pus
introdus: cine
```

Șirurile de caractere însă sunt în realitate pointeri constanți. Astfel, când încercăm să obținem un șir de caractere, nu transmitem o referință către variabila care va conține șirul ci direct variabila (ea este deja referință).

scanf3.c

```c
int main()
{
  char str[100];
  printf("introduceti o expresie:");
   scanf("%[^\n]s", cuvant);
  printf("introdus: %s",str);
  return 0;
}
```

```
introduceti o expresie:cine a pus
introdus: cine a pus
```

Declarată ca în codul de mai sus, scanf va citi până va întâlni primul caracter cu valoare de delimitator (spațiu, tab). Dacă dorim să introducem un șir de caractere care conține spații, specificatorul %s va trebui modificat astfel:

```c
scanf("%[^\n]s", cuvant);
```

Aceasta va determina scanf să se oprească doar la primirea caracterului return (\n).

buffer overrun cu scanf

Orice funcție care conduce la copierea unui șir de caractere într-o anumită variabilă este susceptibilă de buffer overrun.

https://onlinegdb.com/HJ-0hJfvO

```c
#include <stdio.h>
#include <string.h>

int main()
{
    char cuvant[2];

    printf("cuvant:");
    scanf("%[^\n]s", cuvant);
    printf("%s\n",cuvant);
    return 0;
}
```

scanf4.c

https://onlinegdb.com/H1GI61Mv_

```c
#include <stdio.h>
#include <string.h>

int main()
{
    char cuvant[2];
    char str[10];
    printf("cuvant:");
    scanf("%[^\n]s", cuvant);
    printf("%s\n",cuvant);
    return 0;
}
```

scanf5.c

```
introduceti un cuvânt:mare
mare
*** stack smashing detected
***: <unknown> terminated
Aborted (core dumped)
```

```
cuvânt:mare
mare
```

```
cuvânt:mare de tot
mare de tot
*** stack smashing detected ***:
terminated
Aborted (core dumped)
```

Simpla definire a unei variabile suplimentare imediat după cuvant[2] a determinat programul sa nu mai dea eroare. Atentie! în continuare programul este în starea de buffer overrun: cuvant[] trebuie sa nu aibă decât două caractere iar la consolă au fost introduse 4. Și cel de-al doilea program poate fi forțat să depășească spațiul din stack dacă se introduce un șir de caracter mai mare.

*** stack smashing detected ***

Stack smashing reprezintă scrierea datelor dincolo de spațiul din stack rezervat de compilator pentru variabilele unei anumite funcții. În primul caz, singura variabilă definită era cuvant[2] iar apoi spațiul dedicat variabilelor funcției main() se termina. Introducerea unui șir mai mare de 2 a determinat încercarea de a suprascrie conținutul memoriei stack care nu aparținea functiei main(). Compilatorul a introdus o metodă de protecție împotriva acestor încercări și programul a fost oprit.

în cel de-al doilea caz, declararea celei de-a doua variabile (str[10]) a mărit spațiul alocat în stack pentru funcția main() cu 10 octeti, astfel încât, deși a avut loc suprascrierea memoriei rezervate variabilei atr (buffer overrun), totusi, nu a fost depășit spațiul rezervat funcției. Dacă introducem un șir de caractere mai lung decât totalitatea spațiului ocupat de variabilele definite în main() (să zicem un șir de 20 de caractere), programul se va opri din nou cu același mesaj.

Protecția poate fi dezactivată (în cazul gcc cu utilizarea opțiunii de compilare -fno-stack-protector). Dacă este dezactivată, programul poate rula corect, sau, la valori foarte mari ale șirului, se va opri cu eroarea segmentation fault care este sesizată direct de nucleul sistemului de operare.

```
gcc -Wall -fno-stack-protector reading_strings3.c -o reading_strings3
```

```
cuvânt:mare
mare
```

```
cuvânt:celmaimarecuvantdinlume
celmaimarecuvantdinlume
Segmentation fault (core dumped)
```

puts și gets

puts și gets sunt funcții speciale pentru citirea, respectiv scriere șirurilor de caractere.

https://onlinegdb.com/ByWhakGwu

```c
#include <stdio.h>

int main()
{

   char cuvant[100];
   puts("introduceti un cuvânt:");
   gets(cuvant);
   puts(cuvant);

   return 0;
}
```

puts_gets.c

Compilarea acestui program produce o serie de warning-uri.

Funcția gets a fost eliminata din standardul C în anul 1999 din cauza pericolului de a introduce vulnerabilități majore în programe prin erori de tip buffer overrun sau stack overflow.

```
reading_strings4.c: In function ‘main’:
reading_strings4.c:9:5: warning: implicit declaration of function ‘gets’;
did you mean ‘fgets’? [ -Wimplicit-function-declaration ]
     gets(cuvant);
     ^~~~
     fgets
/usr/bin/ld: /tmp/ccU7ge6a.o: in function `main':
reading_strings4.c:(.text+0x30): warning: the `gets' function is dangerous
and should not be used.
```

Citirea șirurilor de la consolă - fgets

https://onlinegdb.com/BJRIRJfDu

```c
#include <stdio.h>
#define MAX 15

int main()
{
    char astr[MAX];
    puts("introduceti un sir:");
    fgets(astr, MAX, stdin);
    printf("string is: %s\n", astr);

    return 0;
}
```

puts_fgets.c

char *fgets(char *str, int n, FILE *stream)

str : Pointer către un șir de caractere unde se va copia ceea ce se citește
n : Numărul maxim de caractere care va fi copiat în str, inclusiv terminatorul nul.
***stream :** Pointer către un fișier deschis (pentru citirea de la consolă se folosește stdin - standard input)

funcția returnează pointerul str

```
introduceti un sir:
vulpea rapida
string is: vulpea rapida
```

```
introduceti un sir:
vulpea rapida sare peste cainele lenes
string is: vulpea rapida
```

Datorită trimiterii numărului de caractere ca parametru de intrare, fgets este o funcție mai sigură decât gets sau scanf. Cunoscând lungimea dorită, fgets poate limita corect numărul de caractere, ocolind depășirea spațiului disponibil.

https://en.wikipedia.org/wiki/The_quick_brown_fox_jumps_over_the_lazy_dog

pointeri către șiruri

https://onlinegdb.com/H1fT1xMP_

În cazul oricărui șir/matrice, variabila definită este un pointer către primul element al acesteia. Astfel, la definirea unui șir

char str[];

str reprezintă un array de char - uri care se va transforma in pointer la trimiterea către o funcție.

Șirurile pot fi definite ca pointeri direct.

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

int main()
{
  char *strp = "caracatita";
  printf ("strp este: %s\n", strp);
  return 0;
}
```

scanf_init.c

https://onlinegdb.com/HJe5Jefv_

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

int main()
{
  char *strp;
  scanf("%s", strp);
  printf ("strp este: %s\n", strp);
  return 0;
}
```

scanf_null.c

```
caracatita
strp este: (null)
```

Un pointer astfel definit NU poate fi folosit în funcția scanf pentru că nu este inițializat. Adica, este alocată memoria pentru a stoca o adresă a altei variabile dar încă nu este nimic stocat în acea memorie.

```
char arry[100];
char *strp = arry;
scanf("%s", strp);
```

Șiruri de șiruri de caractere

```c
#include <stdio.h>

int main() {
 char cuvinte[3][50] =
     {"zero", "one", "two"};

 for (int i = 0; i < 3; i++) {
   printf("%s\n", cuvinte[i]);
 }

 return 0;
}
```

string_of_strings.c

https://onlinegdb.com/H1bS7av94

```c
#include <stdio.h>

int main() {
 char * cuvinte[] =
     {"zero", "one", "two"};

 for (int i = 0; i < 3; i++) {
   printf("%s\n", cuvinte[i]);
 }

 return 0;
}
```

```c
#include <stdio.h>

int main() {
 char cuvinte[][50] =
     {"zero", "one", "two"};

 for (int i = 0; i < 3; i++) {
   printf("%s\n", cuvinte[i]);
 }

 return 0;
}
```

Compilatorul poate calcula mărimea primei dimensiuni a matricii pe baza inițializării acesteia.

Evaluând construcția {“zero”, “one”, “two”}

Inițializarea șirurilor de șiruri de caractere și strcpy

`char cuvinte[3][50]` initializeaza toata matricea corect, asteptand 3 siruri de caractere care pot avea pana la 50 de caractere (49 + null)

strcpy1.c

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

int main(int argc, char **argv) {
 char cuvinte[3][50];

 strcpy(cuvinte[0], "zero");
 strcpy(cuvinte[1], "one");
 strcpy(cuvinte[2], "two");

 for (int i = 0; i < 3; i++) {
   printf("%s\n", cuvinte[i]);
 }
 return 0;
}
```

https://onlinegdb.com/HkT_befD_

```
zero
one
two
```

`char * cuvinte[3]` initializeaza o matrice de 3 pointeri, `cuvinte[n]` = “sir de caractere” initializeaza sirurile

strcpy2.c

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

int main(int argc, char **argv)
{
 char * cuvinte[3];

 cuvinte[0] = "zero";
 cuvinte[1] = "one";
 cuvinte[2] = "two";

 for (int i = 0; i < 3; i++) {
   printf("%s\n", cuvinte[i]);
 }
 return 0;
}
```

declarația cuvinte[] inițializeaza șirurile de caractere

https://onlinegdb.com/SJiNGgfw_

```
zero
one
two
```

Atenție la inițializări !

strcpy_segfault.c

```c
#include <stdio.h>
#include <string.h>

int main() {

char * cuvinte[3];

 strcpy(cuvinte[0], "zero");
 strcpy(cuvinte[1], "one");
 strcpy(cuvinte[2], "two");

 for (int i = 0; i < 3; i++) {
   printf("%s\n", cuvinte[i]);
 }
 return 0;
}
```

https://onlinegdb.com/SJO57efwO

```
Segmentation fault (core dumped)
```

char * cuvinte[3] declară o matrice cu trei pointeri. Dar atat!. Nu se rezervă decât spațiul de memorie necesar pentru pointer. Nu se inițializează spațiul pentru șirul de caractere.

În cazul șirurilor mici, programul poate să pară că funcționează câteodată (suprascriind alte variabile) dar de obicei rezultatul este “segmentation fault”.

Segmentation fault este o veste buna aici, pentru că se poate identifica eroarea.

Toate cele trei instrucțiuni strcpy conduc la “Buffer overrun”.

strcpy așteaptă ca șirul de destinație să fie inițializat și să aibă suficient loc!

dereferențiere multiplă în cazul șirurilor

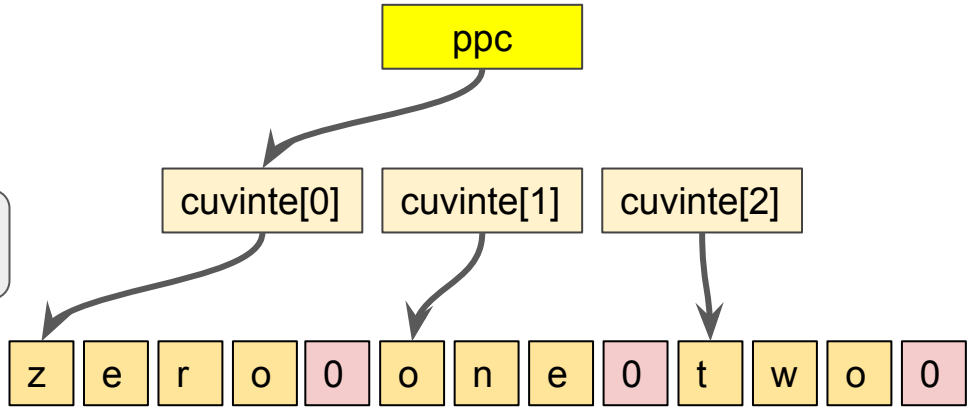
multiple_deref.c

```c
#include <stdio.h>

int main(int argc, char **argv) {
  char *cuvinte[3];
  char *pc;
  char **ppc;

  cuvinte[0] = "zero";
  cuvinte[1] = "one";
  cuvinte[2] = "two";

  ppc = cuvinte;
  for (int i = 0; i < 3; i++) {
     ppc = cuvinte + i;
     pc = *ppc;
     while (*pc != 0) {
        printf("%c ", *pc);
        pc = pc + 1;
     }
     printf("\n");
  }
  return 0;
}
```

matrice de pointeri

https://onlinegdb.com/ByW3NTPqE

```
z e r o
o n e
t w o
```



ppc conține adresa primului element al matricii cuvinte.

matricea cuvinte conține adresele primului element al matricii cuvinte[0], al primului element al matricii cuvinte[1] și al primului element al matricii cuvinte[2],

În bucla for, se ciclează prin matricea cuvinte[] folosindu-se aritmetica pointerului ppc (prin adăugarea lui 1, pointerul ppc se “mută” la următoarea adresă).

În bucla while se cicleaza prin matricile cuvinte[i] folosindu-se adresarea cu pointerul pc, până când caracterul este terminatorul nul

Compararea șirurilor de caractere

int strcmp(const char *str1, const char *str2)

strcmp1.c

```c
#include <stdio.h>
#include <string.h>

int main () {
  char str1[20];
  char str2[20];
  int result;

  strcpy(str1, "caracatita");
  strcpy(str2, "CARACATITA");

  result = strcmp(str1, str2);

  if(result > 0) {
     printf("str1 > str2 (result: %d)\n", result);
  } else if(result < 0) {
     printf("str1 < str2 (result: %d)\n", result);
  } else {
     printf("str1 == str2 (result: %d)\n", result);
  }
  return 0;
}
```

https://onlinegdb.com/r1aVHpD5N

strcmp returnează prima diferență nenulă dintre două caractere aflate pe aceeași poziție în cele două șiruri de caractere

65	41	101 A	A
66	42	102 B	B
67	43	103 C	C
68	44	104 D	D
69	45	105 E	E
70	46	106 F	F
71	47	107 G	G

97	61	141 a	a
98	62	142 b	b
99	63	143 c	c
100	64	144 d	d
101	65	145 e	e
102	66	146 f	f
103	67	147 g	g

99 - 67 = 32

```
str1 > str2 (result: 32)
```

`char * strcat(char *dest, const char *src)`

strcat e o funcție care concatenează cel de-al doilea șir la capătul primului.

strcat1.c

```c
#include <stdio.h>
#include <string.h>

int main() {
  char str1[100] = "If it bleeds, ";
  char str2[] = "we can kill it.";

  strcat(str1, str2);

  printf ("%s\n", str1);

  return 0;
}
```

https://onlinegdb.com/rJBhTmyPO

If it bleeds, we can kill it.

Funcția strcat poate fi apelată fără a se solicita returnarea într-o anumită variabilă.

Dacă este apelată cu returnare într-o variabilă (care trebuie să fie de tip pointer către char) acea variabilă va primi un pointer către primul argument al funcției.

strcat2.c

```c
#include <stdio.h>
#include <string.h>

int main() {
  char str1[100] = "If it bleeds, ";
  char str2[] = "we can kill it.";

  char * str3;
  str3 = strcat(str1, str2);

  printf ("%s\n", str3);

  printf ("%p\n", str1);
  printf ("%p\n", str3);

  return 0;
}
```

https://onlinegdb.com/HyyrTX1Du

If it bleeds, we can kill it.
0x7ffd78a155f0
0x7ffd78a155f0

Implementare strcat

mystrcat.c

```c
#include <stdio.h>

char *mystrcat(char *dest, const char *src)
{
   int index_dest = 0, dest_len = 0, src_len = 0, index_src=0;

   while (dest[index_dest++]) {
       dest_len++;
   }

   while (src[index_src++]) {
       src_len++;
   }

   for (index_dest = 0; index_dest < src_len; index_dest++) {
       dest[dest_len] = src[index_dest];
       dest_len++;
   }

   return dest;
}

int main() {
 char str1[100] = "If it bleeds, ";
 char str2[] = "we can kill it.";

 mystrcat(str1, str2);
 printf ("%s\n", str1);

 return 0;
}
```

https://onlinegdb.com/By9_rgGvd

Numărăm câte caractere are șirul dest

Numărăm câte caractere are șirul src

Copiem caracterele din șirul src (de la 0 până la src_len) în șirul destinație (de la dest_len până la dest_len + src_len)

```
If it bleeds, we can kill it.
```

`char *strtok(char *str, const char *delim)`

```c
#include <string.h>
#include <stdio.h>

int main () {
  char str[80] = "Cine a pus ulei pe clanta";
  const char s[2] = " ";
  char *token;

  token = strtok(str, s);
  printf( "%s\n", token );

  while( token != NULL ) {
     token = strtok(NULL, s);
     if (token != NULL) {
       printf( "%s\n", token );
     }
  }

  return(0);
}
```

https://onlinegdb.com/SkBCUefv_

strtok se apelează de mai multe ori.

Primul apel trimite catre strtok un pointer către șirul care trebuie separat si un pointer către un șir care va fi folosit pe post de separator.

Următoarele apeluri vor trimite doar separatorul, primind un pointer către fiecare fragment.

```
Cine
a
pus
ulei
pe
clanta
```

`char *strtok(char *str, const char *delim)`

strtok2.c

```c
#include <string.h>
#include <stdio.h>

int main () {
  char str[80] = "Cine a pus ulei pe clanta";
  const char s[2] = " ";
  const char a[2] = "a";
  char *token;

  token = strtok(str, s);
  printf( "%s\n", token );

  while( token != NULL ) {
     token = strtok(NULL, s);
     if (token != NULL) {
       printf( "%s\n", token );
     }
  }
  return(0);
}
```

https://onlinegdb.com/BJXOPlMwO

```
Cine
 pus ulei pe cl
nt
```

Apelurile succesive ale funcției strtok pot trimite pointeri către separatori diferiți.

Primul apel a trimis separatorul “ “ (spațiu)

Următoarele apeluri trimit separatorul “a”

`char *strstr ( const char *, const char * );`

strstr.c

```c
#include <stdio.h>
#include <string.h>

int main () {
  char sir[26] = "Cine a pus ulei pe clanta";
  char caut[5] = "ulei";
  char *ret;
  ret = strstr(sir, caut);
  printf("%s\n", ret);
  return(0);
}
```

https://onlinegdb.com/SyNu_IMDO

```
ulei pe clanta
```

Returnează un pointer către prima apariție a șirului “caut” în șirul “șir”

printf -ul apelat cu acest pointer, va printa caracterele începând de la primul caracter către care trimite pointerul returnat până la primul terminator NULL întâlnit (la finalul șirului “sir”.

`char *strchr(const char *str, int ch)`

strchr.c

```c
#include <stdio.h>
#include <string.h>

int main () {
  char sir[] = "Cine a pus ulei pe clanta";
  char caut = 'u';
  char *ret;
  ret = strchr(sir, caut);
  printf("%s", ret);
  return 0;
}
```

https://onlinegdb.com/ryxhulfvd

```
us ulei pe clanta
```

Returnează un pointer către prima apariție a caracterului din variabila “caut” în șirul “șir”.

printf -ul apelat cu acest pointer, va printa caracterele începând de la primul caracter către care trimite pointerul returnat până la primul terminator NULL intâlnit (la finalul șirului “sir”.

Funcții de conversie

atof	double atof(const char *str) Convertește sirul de octeți str intr-un număr flotant de tip double
atoi atol atoll	int atoi(const char *str) long int atol(const char *str) long long int atoll(const char* str); Convertește șirul de octeți str într-un număr întreg de tip int, long sau long long
strtof strtod	float strtof(const char* str, char **endptr) double strtod(const char *str, char **endptr) Convertește șirul de caractere str într-un număr flotant, de tip float sau double.
strtol strtoll strtoul strtoull	long int strtol(const char *str, char **endptr, int base) Convertește sirul de caractere str intr-un număr întreg de tip long, long long, unsigned long sau unsigned long long

strtof, strtod

strtof.c

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

int main () {
  char str[30] = "20.30300 si ceva litere";
  char *ptr;
  double ret;

  ret = strtof(str, &ptr);
  printf("Numarul flotant este %f\n", ret);
  printf("Restul sirului:  %s", ptr);

  return(0);
}
```

https://onlinegdb.com/S19DmZzwO

```
Numarul flotant este 20.302999
Restul sirului:   si ceva litere
```

șirul care va fi analizat

pointer cu ajutorul căruia se poate obține restul șirului

variabila în care se va returna numărul flotant.

Cele două funcții au același comportament, ele diferă doar prin tipul de dată flotantă pe care îl returnează.

strtof și strtod sunt capabile să “extragă” numărul flotant aflat la începutul unui șir de caractere, returnând în același timp și restul șirului.

atoi, atol, atoll

atoi.c

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

int main () {
  int val, aal;
  char str[20] = "98993489";
  char atr[20] = "astalavista";

  val = atoi(str);
  printf("sirul este: %s, valoarea este: %d\n", str, val);

  aal = atoi(atr);
  printf("sirul este: %s, valoarea este:  %d\n", atr, aal);

  return(0);
}
```

https://onlinegdb.com/rykGPbfwu

```
sirul este: 98993489, valoarea este: 98993489
sirul este: astalavista, valoarea este:  0
```

conversie numeric - string

```c
int sprintf(char *str, const char *format, ...)
```

sprintf.c

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

int main () {
  char str[80];

  sprintf(str, "%f", M_PI);
  printf("Valoarea lui pi este: %s", str);

  return(0);
}
```

https://onlinegdb.com/Hk7jqbfwd

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	Start of Header	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	Start of Text	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	End of Text	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	End of Transmission	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enquiry	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Acknowledgment	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Backspace	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	Horizontal Tab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	Line feed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	Vertical Tab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	Form feed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	Carriage return	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	Shift Out	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	Shift In	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	Data Link Escape	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	Device Control 1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	Device Control 2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	Device Control 3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	Device Control 4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	Negative Ack.	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Synchronous idle	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	End of Trans. Block	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Cancel	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	End of Medium	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Substitute	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Escape	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	File Separator	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	Group Separator	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	Record Separator	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	Unit Separator	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	Del

asciichars.com

Fișiere

Lucrul cu fișiere

Un fișier este o colecție identificabilă de date care este stocată într-un sistem care nu permite accesul direct.

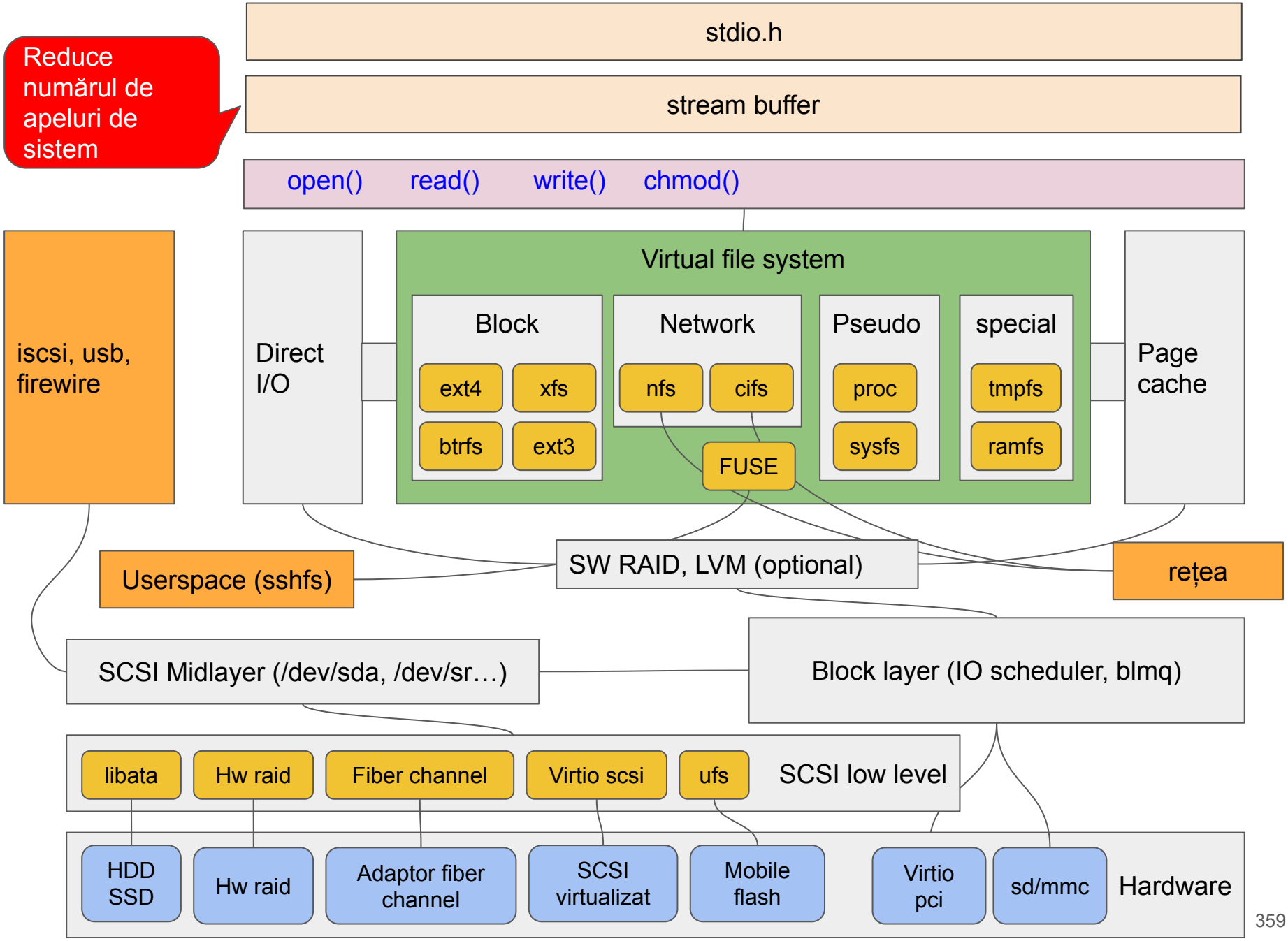
Stocarea și accesul la aceste date se face cu ajutorul unui sistem de organizare a informației (o structură complicată de pointeri) care se numește “Sistem de fișiere”. Sistemul de fișiere permite identificarea (de obicei pe baza unei structuri ierarhice) și accesarea unui fișier.

Pentru a putea opera cu datele stocate într-un fișier este necesară aducerea lor în memorie. în funcție de nevoi, datele stocate într-un fișier se pot aduce în memorie:

- complet sau parțial
- într-una sau în mai multe variabile
- prin citire secvențială sau aleatoare

Exista patru tipuri de funcții importante la manevrarea fișierelor:

- Functii de deschidere / inchidere a fisierului
- Functii de citire din stream
- Functii de scriere in stream
- Functii de pozitionare in stream


Reduce numărul de apeluri de sistem
stdio.h
stream buffer
open()
read()
write()
chmod()
iscsi, usb, firewire
Direct I/O
Virtual file system
Block
ext4
xfs
btrfs
ext3
Network
nfs
cifs
FUSE
Pseudo
proc
sysfs
special
tmpfs
ramfs
Page cache
Userspace (sshfs)
SW RAID, LVM (optional)
rețea
SCSI Midlayer (/dev/sda, /dev/sr…)
Block layer (IO scheduler, blmq)
libata
Hw raid
Fiber channel
Virtio scsi
ufs
SCSI low level
HDD SSD
Hw raid
Adaptor fiber channel
SCSI virtualizat
Mobile flash
Virtio pci
sd/mmc
Hardware

flux ASCII (text) sau binar.

Comenzile de interacțiune cu fișierele NU interacționează niciodată direct cu mediul de stocare, ci cu reprezentarea datelor în memoria computerului. Această reprezentare este frecvent privită ca fiind un flux de octeți (stream). Cuvântul stream apare frecvent în descrierea interacțiunilor cu fișiere, de altfel, în C++ se adaugă un nou sistem de interacțiune cu fișierele, numit iostream (input/output stream). De regulă accesul la flux se face în mod text sau în mod binar.

Modul de acces la flux (text sau binar) se specifică de la deschiderea fișierului.

În general, fișierele text (cele care conțin un sir de caractere ASCII) sunt tratate ca un șir de caractere care se termina la EOF.

Fișierele binare sunt tratate ca niște structuri de date în care se cunosc exact pozițiile și tipul de date stocate.

Citirea unui fișier text înseamnă de cele mai multe ori absorbția în memorie a întregului șir de octeți până la apariția caracterului EOF, fără a avea nici o altă informație structurală. Aceasta este citirea secvențială.

Citirea unui fișier binar înseamnă de cele mai multe ori o suită de perechi de operații de poziționare la anumite adrese urmate de instrucțiuni de citire a unui număr precis de octeți și plasarea lor în variabile de tipuri diferite. Aceasta se numește citirea aleatoare (random).

Funcții pentru operații cu fișiere

Interactiune cu fisierle	Citire din fisiere	Scriere in fisiere	Navigare prin fisiere
fopen() fclose()	fread() fscanf() fgetc() fgets()	fwrite() fprintf() fputc() fputs()	fseek() ftell() rewind()

Scriere și citire din fișiere

Citire din fișiere	Scriere in fisiere
fread() size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) Citește un array de count elemente, fiecare cu dimensiunea size (în octeți) și le stochează într-un bloc de memorie specificat de pointerul ptr.	**fwrite()** size_t fwrite (const void * ptr, size_t size, size_t count, FILE * stream); Scrie un array de count elemente, fiecare cu dimensiunea size în octeți, din blocul de memorie specificat de pointerul ptr în poziția curentă a fluxului.
fscanf() int fscanf (FILE * stream, const char * format, ...); Citește date din flux conform parametrului format și le stochează în pointeri predefiniți și specificați în argumentele funcției	**fprintf()** int fprintf (FILE * stream, const char * format, ...); Scrie în flux șirul comnstruit pe baza argumentului “format” și a variabilelor transmise. Similar cu printf
fgetc() int fgetc (FILE * stream); Returnează caracterul (octetul) care se află la poziția curentă a fluxului. Apoi, pointerul intern al fluxului este mutat la următoarea poziție	**fputc()** int fputc (int character, FILE * stream); Scrie un *octet (character)* în *stream* și mută poziția pointerului cu un octet.
fgets() char * fgets (char * str, int num, FILE * stream); Citește un număr arbitrar de caractere din stream până se ajunge la num-1 octeți, un caracter de tip newline sau sfârșitul fișierului.	**fputs()** int fputs (const char * str, FILE * stream); Scrie șirul către care trimite pointerul str în stream, începând de la poziția curentă.

structura opacă _IO_FILE linux (pentru fiecare sistem de operare este diferită)

```
struct _IO_FILE
{
 int  flags;/* High-order word is  IO MAGIC; rest is flags. */
 char * IO read ptr;/* Current read pointer */
 char * IO read end;/* End of get area. */
 char * IO read base;/* Start of putback+get area. */
 char * IO write base;/* Start of put area. */
 char * IO write ptr;/* Current put pointer. */
 char * IO write end;/* End of put area. */
 char * IO buf base;/* Start of reserve area. */
 char *_IO_buf_end;/* End of reserve area. */

 /* The following fields are used to support backing up and undo. */
 char * IO save base;
 char * IO backup base;
 char *_IO_save_end;

 struct  IO marker * markers;
 struct  IO FILE *_chain;
 int  fileno;
 int  flags2;
 unsigned short  cur column;
 signed char  vtable_offset;
 char  shortbuf[1];
  IO_lock_t *_lock;
};
```

Pointeri către stream buffer

Lista înlănțuită cu legături cu streamurile standard

File handle, returnat de sistemul de operare

fopen

- Alocă structura FILE
  - Inițializează structura FILE
    - Deschide fișierul

fread

- Alocă buffer-ul dacă este NULL
  - Citește datele în buffer
    - Copiază datele din buffer în variabila destinație

fwrite

- Alocă buffer-ul dacă este NULL
  - Copiază datele din variabilă în buffer
    - Trimite datele din buffer către sistemul de operare

fclose

- Deconectează structura FILE
  - Golește și eliberează bufferul
    - închide fișierul
      - Eliberează memoria ocupată de structura FILE

```
FILE *fopen(const char *filename, const char *mode)
```

Opțiuni de deschidere a fișierului în mod text (ASCII)

Mod		Descriere	Dacă fișierul există	Dacă fișierul nu există
r	citire	Deschide pentru citire	Citește de la început	fopen() returnează NULL
w	scriere	Deschide pentru scriere	Suprascrie conținutul	Fișierul va fi creat gol
a	adăugare	Deschide pentru adăugare, datele sunt adăugate la finalul fișierului	Adaugă la conținut	Fișierul va fi creat gol
r+	citire extinsă	Deschide pentru citire și scriere	Citește de la început	Eroare
w+	scriere extinsă	Crează pentru citire și scriere	Suprascrie conținutul	Fișierul va fi creat gol
a+	adăugare extinsă	Deschide pentru citire și adăugare	Adaugă la finalul fișierului	Fișierul va fi creat gol

```
fp = fopen ("file.txt", "w+");
```

```
FILE *fopen(const char *filename, const char *mode)
```

Opțiuni de deschidere a fișierului în mod binar

Mod		Descriere	Dacă fișierul există	Dacă fișierul nu există
rb	citire	Deschide pentru citire	Citește de la început	fopen() returnează NULL
wb	scriere	Deschide pentru scriere	Suprascrie conținutul	Fișierul va fi creat gol
ab	adăugare	Deschide pentru adăugare, datele sunt adăugate la finalul fișierului	Adaugă la conținut	Fișierul va fi creat gol
rb+	citire extinsă	Deschide pentru citire și scriere	Citește de la început	Eroare
wb+	scriere extinsă	Crează pentru citire și scriere	Suprascrie conținutul	Fișierul va fi creat gol
ab+	adăugare extinsă	Deschide pentru citire și adăugare	Adaugă la finalul fișierului	Fișierul va fi creat gol

Stocarea într-un fișier text a unui număr întreg (ex: 1234) se face cu ajutorul a 4 caractere ASCII, 1, 2, 3, 4 (adica 00000001, 00000010, 00000011, 00000100 - 4 octeți). Dar, în același timp, reprezentat ca număr întreg, 1234 este 0000010011010010 care poate fi stocat pe doar 2 octeți,

Numerele sunt stocate mult mai eficient în format binar direct. Pentru a le putea trimite în fișier și a le putea citi din fișier este nevoie ca pointerii să se mute corespunzător, conform variabilelor citite. Aceasta se poate face dacă fișierele se deschid în mod binar.

Nimic nu ne împiedică să citim, octet cu octet, un fișier ASCII ca fișier binar, inspectând fiecare pattern citit și interpretându-l ca atare.

Scrierea în fișier (putc)

```
int putc ( int character, FILE * stream );
```

file_write_pointers.c

```c
#include <stdio.h>

int main()
{
 FILE *fp;
 int c = 'a';
 fp = fopen("test.txt", "w");
 if (fp) {
     printf("read_ptr: %p\n", fp->_IO_read_ptr);
     printf("read_ptr: %p\n", fp->_IO_write_ptr);
     c = putc(c,fp);
     printf("Char: %c\n", c);
     printf("read_ptr addr: %p\n", fp->_IO_read_ptr);
     printf("write_ptr: %p\n", fp->_IO_write_ptr);
     printf("read_ptr: %c\n", *(fp->_IO_read_ptr));
     c = putc(c,fp);
     printf("Char: %c\n", c);
     printf("read_ptr addr: %p\n", fp->_IO_read_ptr);
     printf("write_ptr: %p\n", fp->_IO_write_ptr);
     printf("read_ptr: %c\n", *(fp->_IO_read_ptr));
     c = putc(c,fp);
     printf("Char: %c\n", c);
     printf("read_ptr addr: %p\n", fp->_IO_read_ptr);
     printf("write_ptr: %p\n", fp->_IO_write_ptr);
     printf("read_ptr: %c\n", *(fp->_IO_read_ptr));

   }
 fclose(fp);
 return 0;
}
```

https://onlinegdb.com/B1JWyfzD_

Cât timp fișierul este deschis, se mențin informații cu privire la poziția curentă în fișier.

pointerul write_ptr se mută odată cu operația de citire
pointerul read_ptr rămâne pe loc

```
read_ptr: (nil)
read_ptr: (nil)
Char: a
read_ptr addr: 0x7fa5e3450000
write_ptr: 0x7fa5e3450001
read_ptr: a
Char: a
read_ptr addr: 0x7fa5e3450000
write_ptr: 0x7fa5e3450002
read_ptr: a
Char: a
read_ptr addr: 0x7fa5e3450000
write_ptr: 0x7fa5e3450003
read_ptr: a
```

Fișierul conține: aaa

Citirea din fișier (getc)

int getc(FILE *stream)

file_read_pointers.c

```c
#include <stdio.h>

int main()
{
 FILE *fp;
 int c;
 fp = fopen("test.txt", "r");

 if (fp) {
     printf("read_ptr: %p\n", fp->_IO_read_ptr);
     printf("read_ptr: %p\n", fp->_IO_write_ptr);
     c = getc(fp);
     printf("Char: %c\n", c);
     printf("read_ptr addr: %p\n", fp->_IO_read_ptr);
     printf("write_ptr: %p\n", fp->_IO_write_ptr);
     printf("read_ptr: %c\n", *(fp->_IO_read_ptr));
     c = getc(fp);
     printf("Char: %c\n", c);
     printf("read_ptr addr: %p\n", fp->_IO_read_ptr);
     printf("write_ptr: %p\n", fp->_IO_write_ptr);
     printf("read_ptr: %c\n", *(fp->_IO_read_ptr));
     c = getc(fp);
     printf("Char: %c\n", c);
     printf("read_ptr addr: %p\n", fp->_IO_read_ptr);
     printf("write_ptr: %p\n", fp->_IO_write_ptr);
     printf("read_ptr: %c\n", *(fp->_IO_read_ptr));

}
 fclose(fp);
 return 0;
}
```

https://onlinegdb.com/SyFnNMGPO

Cât timp fișierul este deschis, se mențin informații cu privire la poziția curentă în fișier.

pointerul read_ptr se mută odată cu operația de citire
pointerul write_ptr rămâne pe loc

```
read_ptr: (nil)
read_ptr: (nil)
Char: a
read_ptr addr: 0x7f93a659f001
write_ptr: 0x7f93a659f000
read_ptr: b
Char: b
read_ptr addr: 0x7f93a659f002
write_ptr: 0x7f93a659f000
read_ptr: c
Char: c
read_ptr addr: 0x7f93a659f003
write_ptr: 0x7f93a659f000
read_ptr:
```

Fișierul conține: abc

Scrierea în fișier (fprintf)

int fprintf (FILE * stream, const char * format, ...);

```
#include <stdio.h>

main()
{
  FILE *fp;

  fp = fopen("test.txt", "w");
  fprintf(fp, "Testing...\n");
  fclose(fp);
}
```

https://onlinegdb.com/B1RLbv1w_

La mai multe rulări succesive, fișierul test.txt va fi suprascris. Fișierul va conține de fiecare dată doar un șir: Testing...

```
#include <stdio.h>

main()
{
  FILE *fp;

  fp = fopen("test.txt", "a");
  fprintf(fp, "Testing...\n");
  fclose(fp);
}
```

https://onlinegdb.com/rkgXt-wyvu

La mai multe rulări succesive, fișierul test.txt va fi deschis iar șirul “Testing…” va fi adăugat.

Pentru fiecare rulare, în fișier se va adăuga un alt rând conținând șirul “Testing…”

Scrierea în fișier (fputs)

int fputs(const char *str, FILE *stream)

```
#include <stdio.h>

main()
{
  FILE *fp;

  fp = fopen("test.txt", "w");
  fputs("Testing...", fp);
  fclose(fp);
}
```

https://onlinegdb.com/ryxfbfzwu

La mai multe rulări succesive, fișierul test.txt va fi suprascris. Fișierul va conține de fiecare dată doar un șir: Testing...

```
#include <stdio.h>

main()
{
  FILE *fp;

  fp = fopen("test.txt", "a");
  fputs("Testing...", fp);
  fclose(fp);
}
```

https://onlinegdb.com/rkgXt-wyvu

La mai multe rulări succesive, fișierul test.txt va fi deschis iar șirul “Testing…” va fi adăugat.

Pentru fiecare rulare, în fișier se va adăuga un alt rând conținând șirul “Testing…”

Citirea dintr-un fișier (fgets)

```
char * fgets ( char * str, int num, FILE * stream );
```

file_fgets.c

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

int main() {
  char buf[1024];
  FILE *fp;
  size_t nread;

  fp = fopen("testlg.txt", "r");
  if (fp) {
      while (fgets(buf, sizeof(buf), fp)) {
       printf("%s", buf);
       }
      fclose(fp);
  }
  return 0;
}
```

https://onlinegdb.com/HJVLXzGwO

```
citit: Cine bate seara
citit: La fereastra mea
```

fgets se oprește la întâlnirea unui caracter de tip EOF sau la sfârșit de rând.

Deschiderea fișierului

Se citeste din fișier o cantitate de octeți egală cu dimensiunea bufferului

Atâta timp cât fgets returnează altceva decât un pointer NULL, se trimite buf la printf.

Citirea unui fișier (fread)

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

file_ascii_read.c

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

int main() {
   char buf[1024];
   FILE *fp;
   size_t nread;

   fp = fopen("test.txt", "r");
   if (fp) {
       while ((nread = fread(buf, 1, sizeof buf, fp)) > 0) {
           printf("read: %ld:  %s\n",nread, buf);
           memset(buf, '\0', sizeof buf);
       }
       fclose(fp);
   }
   return 0;
}
```

https://onlinegdb.com/rJvqRP1vd

Deschiderea fișierului

Se citeste din fișier o cantitate de octeți egală cu dimensiunea bufferului

Se golește bufferul pentru a se evita cazul în care ceea ce se citește în buffer este mai mic decât ceea ce a fost citit anterior.

Dimensiunea bufferului determină numărul citirilor

read: 43: Primul rand Al doilea rand Al treilea rand	`char buf[1024]`
read: 24: Primul rand Al doilea ra read: 19: nd Al treilea rand	`char buf[24]`

Citirea unui fișier cu dimensiune necunoscută

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

int main()
{

    FILE *fp;
    fp = fopen("bigf.txt", "r");

    fseek(fp, 0, SEEK_END);

    long fsize = ftell(fp);

    fseek(fp, 0, SEEK_SET);   /* same as rewind(f); */

    printf("File size: %ld", fsize);

    char *string = malloc(fsize + 1);
    fread(string, 1, fsize, fp);
    fclose(fp);

    string[fsize] = 0;

    printf ("%s", string);

    return 0;
}
```

Deschidem fișierul

Mută pointerul `read_ptr` acolo unde este finalul fișierului (SEEK_END)

ftell returnează poziția curentă a pointerului. Fiind la final, vom obține numărul de octeți.

Mutăm pointerul la început

Alocăm cantitatea necesară de memorie

Citim întreg conținutul fișierului în string

Adăugăm terminatorul NULL

Citirea unui fișier binar

Fișierele cu extensia *.mp3 sunt fișiere în care se stochează semnal audio, comprimat cu un algoritm de compresie aproximativ. Algoritmul de compresie nu păstrază toată informația din semnalul sonor ci doar acea parte care are maximum de efect în experiența ascultătorului.

Algoritmul este bazat pe ceea ce se numește model psihoacustic un model fizic al felului în care creierul uman percepe sunete.

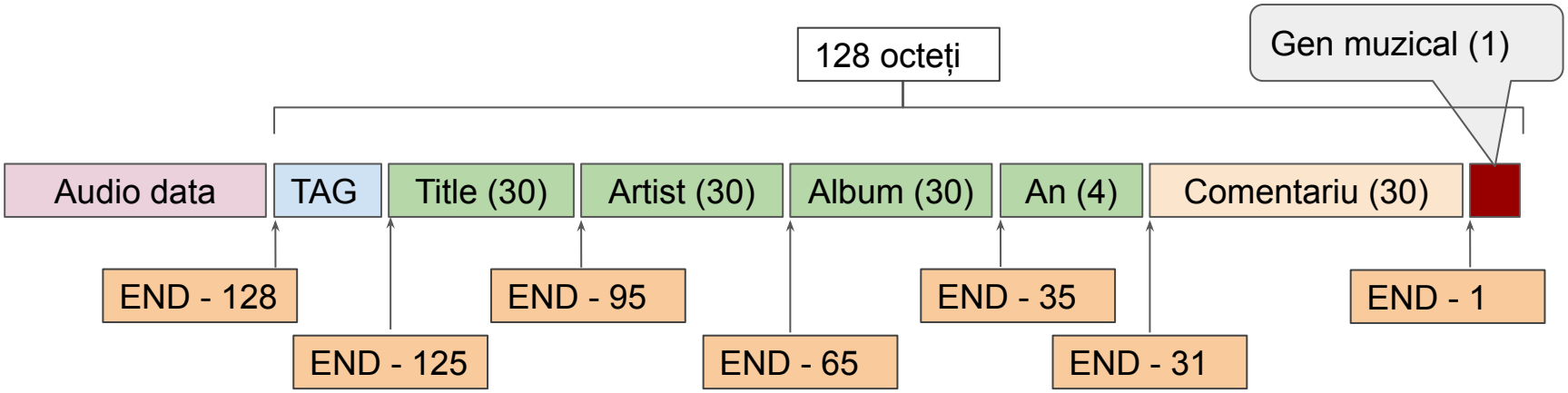
Pe langa streamul audio, un fișier mp3 conține si o serie de informații despre melodie: autor, titlu, etc. Acestea se află într-o structură de date compactă aflată la sfârșitul fișierului.

Există mai multe standarde pentru stocarea acestor informații, numite generic TAG-uri ID3.

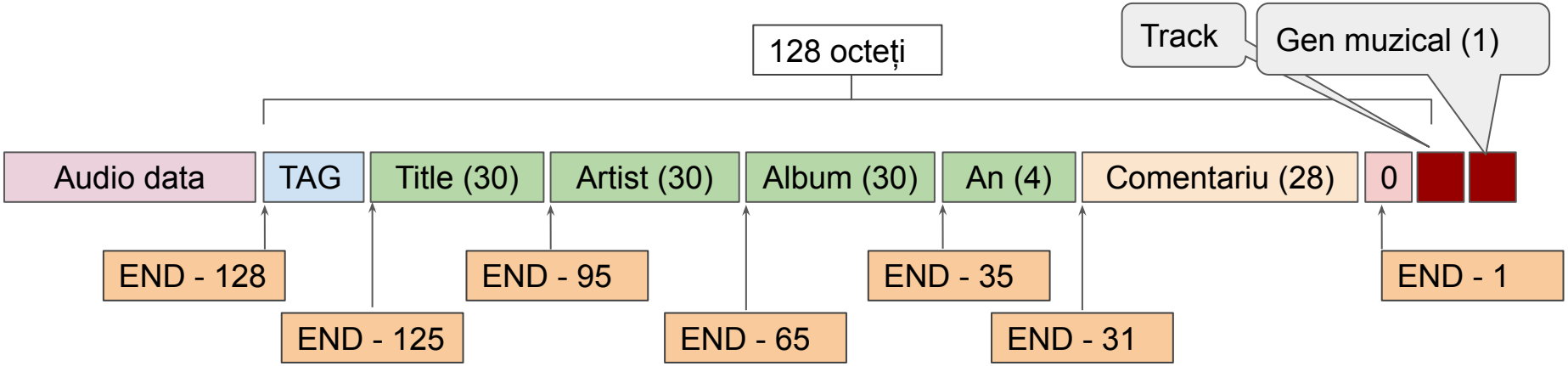
Citirea informațiilor din zona ID3 se poate face doar cunoscând perfect structura acestuia (câți octeți are fiecare câmp și unde începe).

Structura informației în tagurile id3

id3 v1 (Eric Kemp, 1996)



id3 v1.1 (Michael Mutschler, 1997)



```c
int fseek(FILE *stream, long int offset, int whence)
```

```c
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

file_mp3_read.c

```c
#include <stdio.h>
#include <string.h>

int main()
{
  char title[31] = "";
  char artist[31] = "";
  char album[31] = "";
  char year[5] = "";
  char track;
  char genre;

 FILE *song_file;
 song_file = fopen("demo.mp3", "rb+");

 fseek(song_file, -125, SEEK_END);
 fread(title, 1, 30, song_file);
 fread(artist, 1, 30, song_file);
 fread(album, 1, 30, song_file);
 fread(year, 1, 4, song_file);
 fread(&track, 1, 1, song_file);
 fread(&genre, 1, 1, song_file);

 printf ("Title: %s\n", title);
 printf ("Artist: %s\n", artist);
 printf ("Album: %s\n", album);
 printf ("year: %s\n", year);
 printf ("track: %i\n", track);
 printf ("genre: %i\n", genre);
}
```

https://onlinegdb.com/rkenyvVgvu

fread citește numărul nmemb de date de dimensiunea size din fișierul din spatele pointerului *stream și le plasează în variabila din spatele pointerului *ptr.

În cazul variabilelor de tip char[] care sunt pointeri constanți transmitem direct variabila, în cazul variabilelor de tip char transmitem referința către variabile (la fel ca la scanf).

Poziționează pointerul de citire la 125 de octeți distanță de finalul fișierului.

Citește 30 de octeți și îi trimite în stringul title

Citește 30 de octeți și îi trimite în stringul artist

Citește 4 octeți și îi trimite în stringul year

Citește 1 octet și îi trimite în referința către track

Citește 1 octet și îi trimite în referința către genre

```
Title: Llama Whippin' Intro
Artist: DJ Mike Llama
Album: DJ Mike Llama
year: 2005
track: 1
genre: 17
```

```c
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

file_mp3_write.c

```c
#include <stdio.h>
#include <string.h>

int main()
{
 char title[31] = "";
 char artist[31] = "";
 char album[31] = "";
 char year[5] = "";
 char track;
 char genre;

 char wtitle[30] = "Fragil";
 char wartist[30] = "Petronel Parizer";

 FILE *song_file;
 song_file = fopen("demo.mp3", "rb+");

 fseek(song_file, -125, SEEK_END);
 fwrite(wtitle, sizeof(wtitle), 1, song_file);
 fwrite(wartist, sizeof(wartist), 1, song_file);

 fclose(song_file);

 song_file = fopen("demo.mp3", "rb+");

 fseek(song_file, -125, SEEK_END);
 fread(title, 1, 30, song_file);
 fread(artist, 1, 30, song_file);
 fread(album, 1, 30, song_file);

 printf ("Title: %s\n", title);
 printf ("Artist: %s\n", artist);
 printf ("Album: %s\n", album);
}
```

Poziționează pointerul de citire la 125 de octeți distanță de finalul fișierului.

Scrie șirul wtitle în fișier

Scrie șirul wartist în fișier

Închide fișierul

```
Title: Fragil
Artist: Petronel Parizer
Album: Beats of Burdon
```

https://onlinegdb.com/ByRLjExDO

Recursivitate

Recursivitate

recurs1.c

```
//gcc -Wall recurs1.c -o recurs1

#include <stdio.h>

long int fact(int n);

int main()
{
  int n = 4;
  printf("Factorial al lui %d = %ld\n", n, fact(n));
  return 0;
}

long int fact(int nm)
{
  if (nm >= 1) {
      return nm*fact(nm-1);
  }
  return 1;
}
```

Funcția “fact” se apelează pe ea însăși.

Fiecare apel se face cu argumentul “nm -1” - oricât ar fi de mare, la fiecare apel argumentul scade cu 1 deci stim precis ca funcția se va opri la un moment dat.

Factorialul lui 0 este 1.

Funcțiile recursive trebuie să se oprească în toate cazurile posibile.

https://onlinegdb.com/S1VuAKWuO

```
Factorial al lui 4 = 24
```

Recursivitate

```
#include <stdio.h>
long int fact(int n);

int main()
{
   int n = 4;
   long int res = fact(n);
   printf("Factorial al lui %d = %ld\n", n, res);
   return 0;
}
long int fact(int nm)
{
   int r, tmp;
   printf ("intrare: %d\n",nm);
   if (nm >= 1) {
       tmp = fact(nm-1);
       r= nm*tmp;
       printf ("iesire: %d\n", r);
       return r;
       }
   else {
       printf ("else: iesire: 1\n");
       return 1;
       }
}
```

recurs2.c

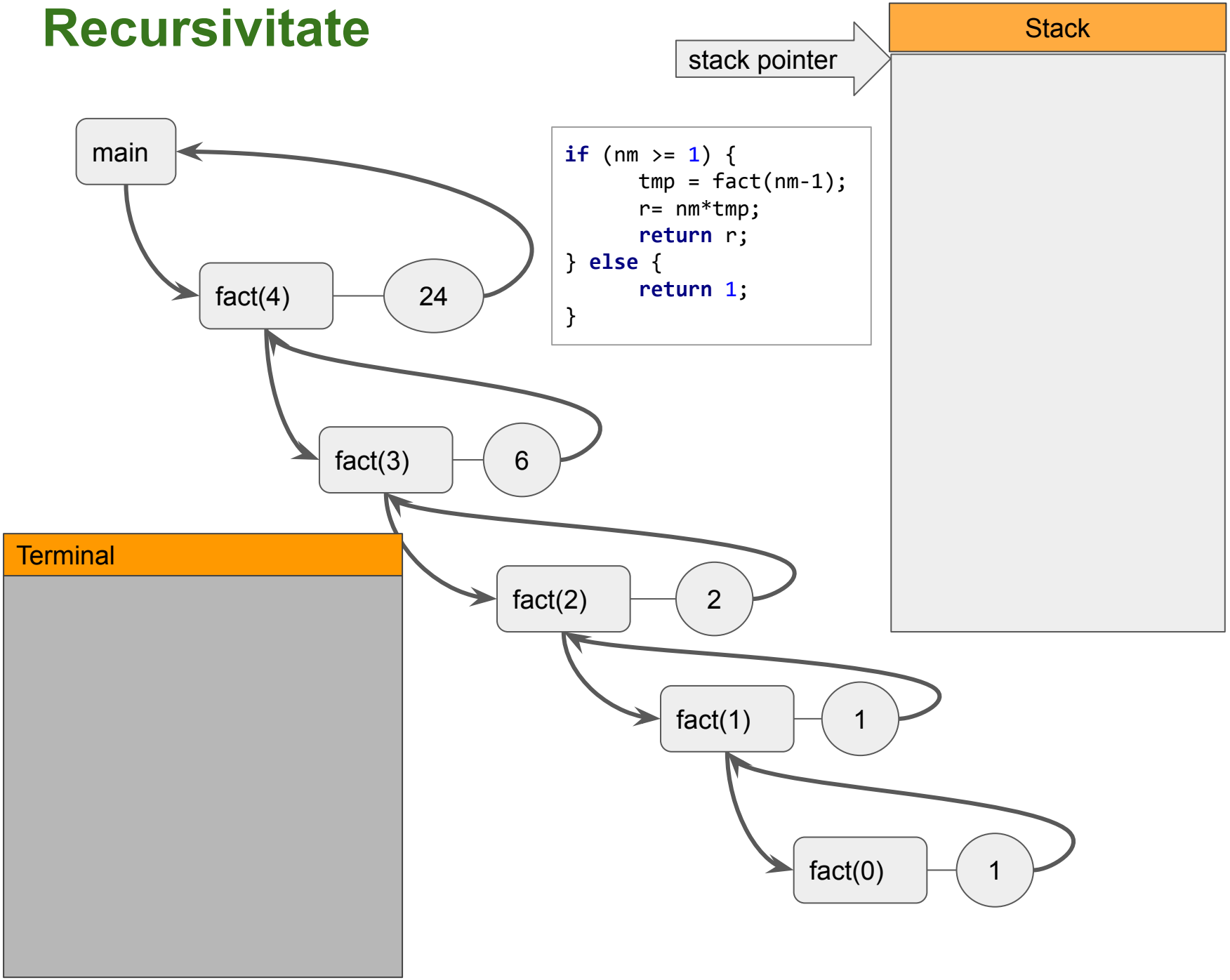
```
incepem...
intrare: 4
intrare: 3
intrare: 2
intrare: 1
intrare: 0
else: iesire: 1
iesire: 1
iesire: 2
iesire: 6
iesire: 24
Factorial al lui 4 = 24
```

după apelul recursiv

înainte de apelul recursiv

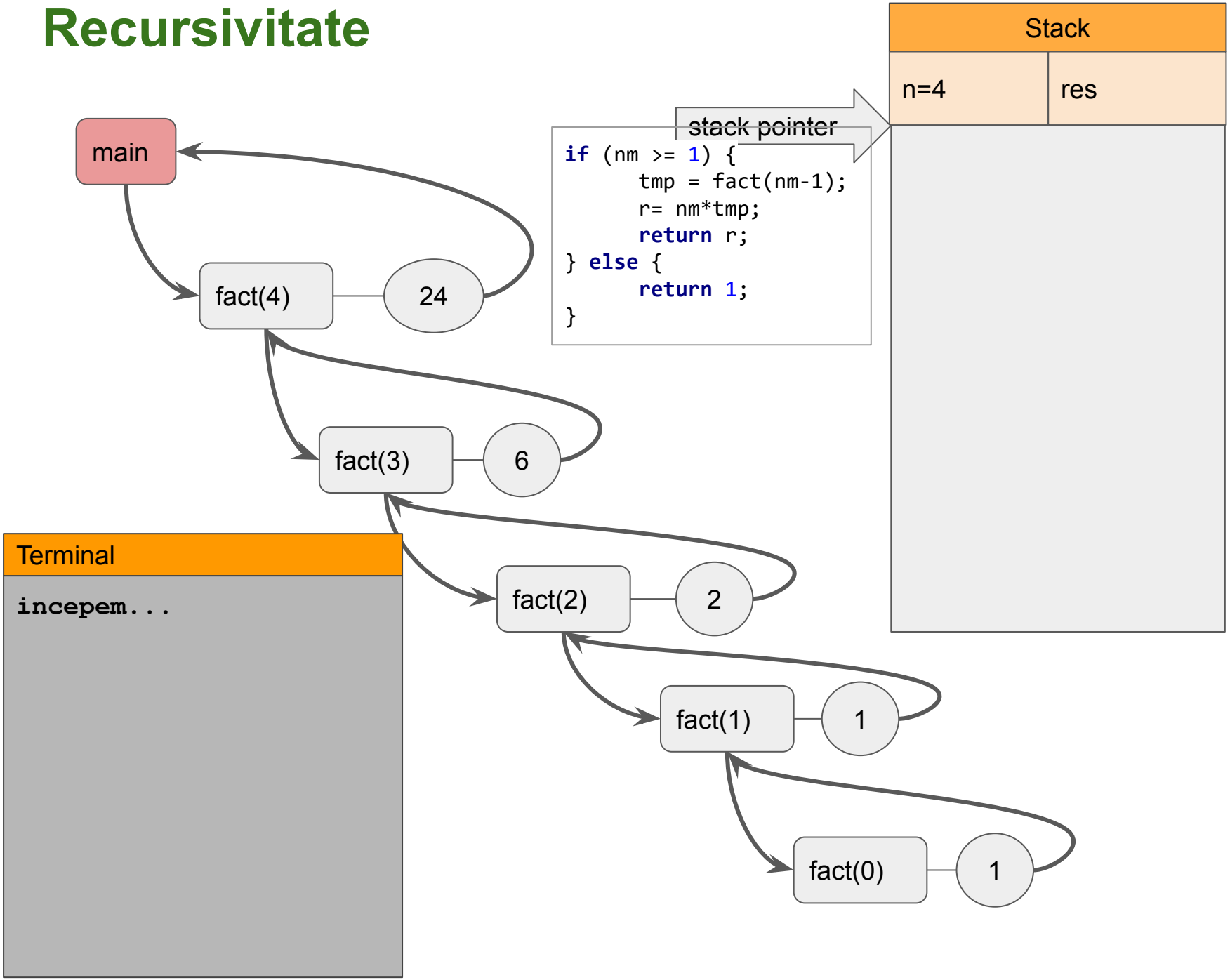
https://onlinegdb.com/uUMdEzIj4

Recursivitate

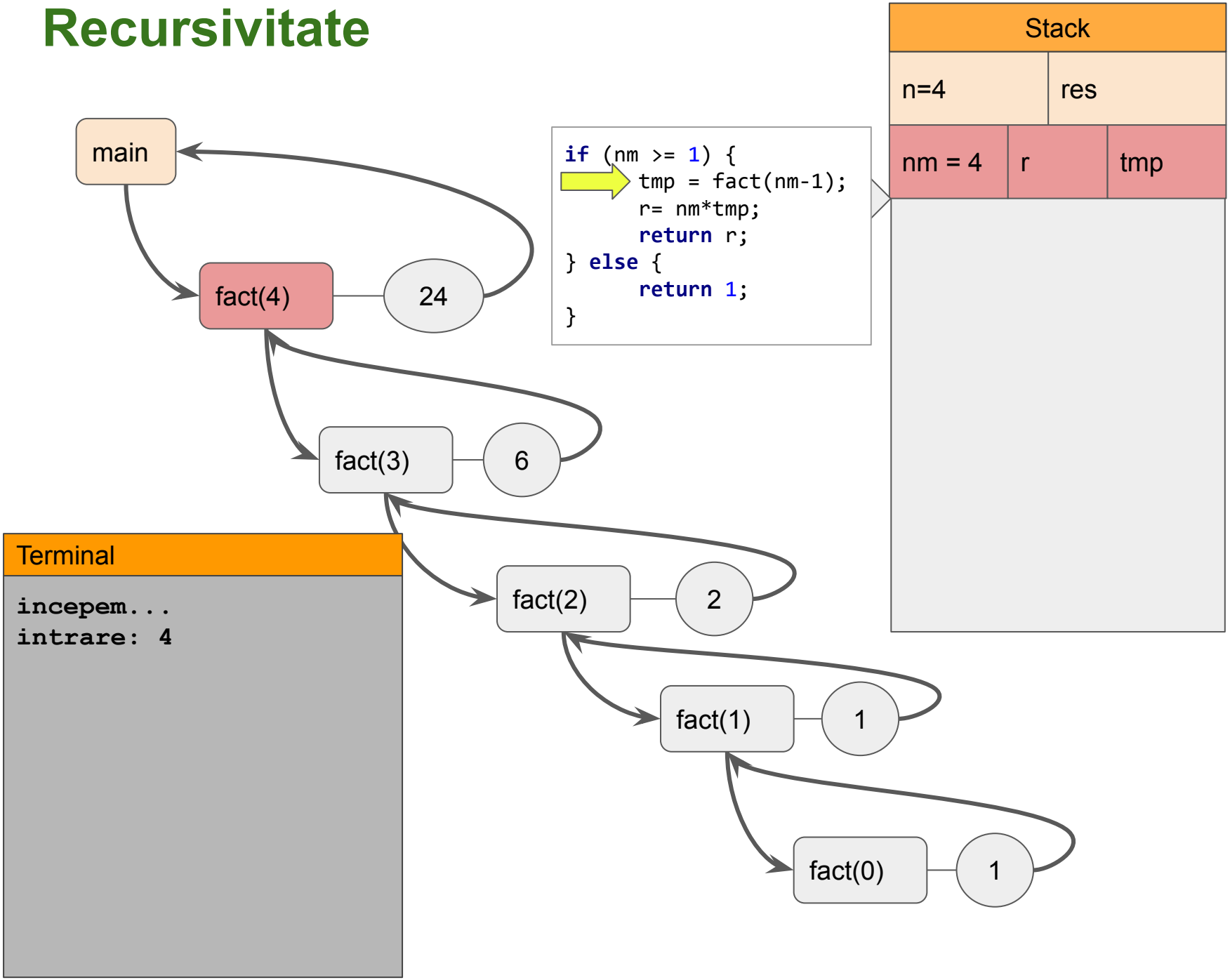



```
if (nm >= 1) {
      tmp = fact(nm-1);
      r= nm*tmp;
      return r;
} else {
      return 1;
}
```

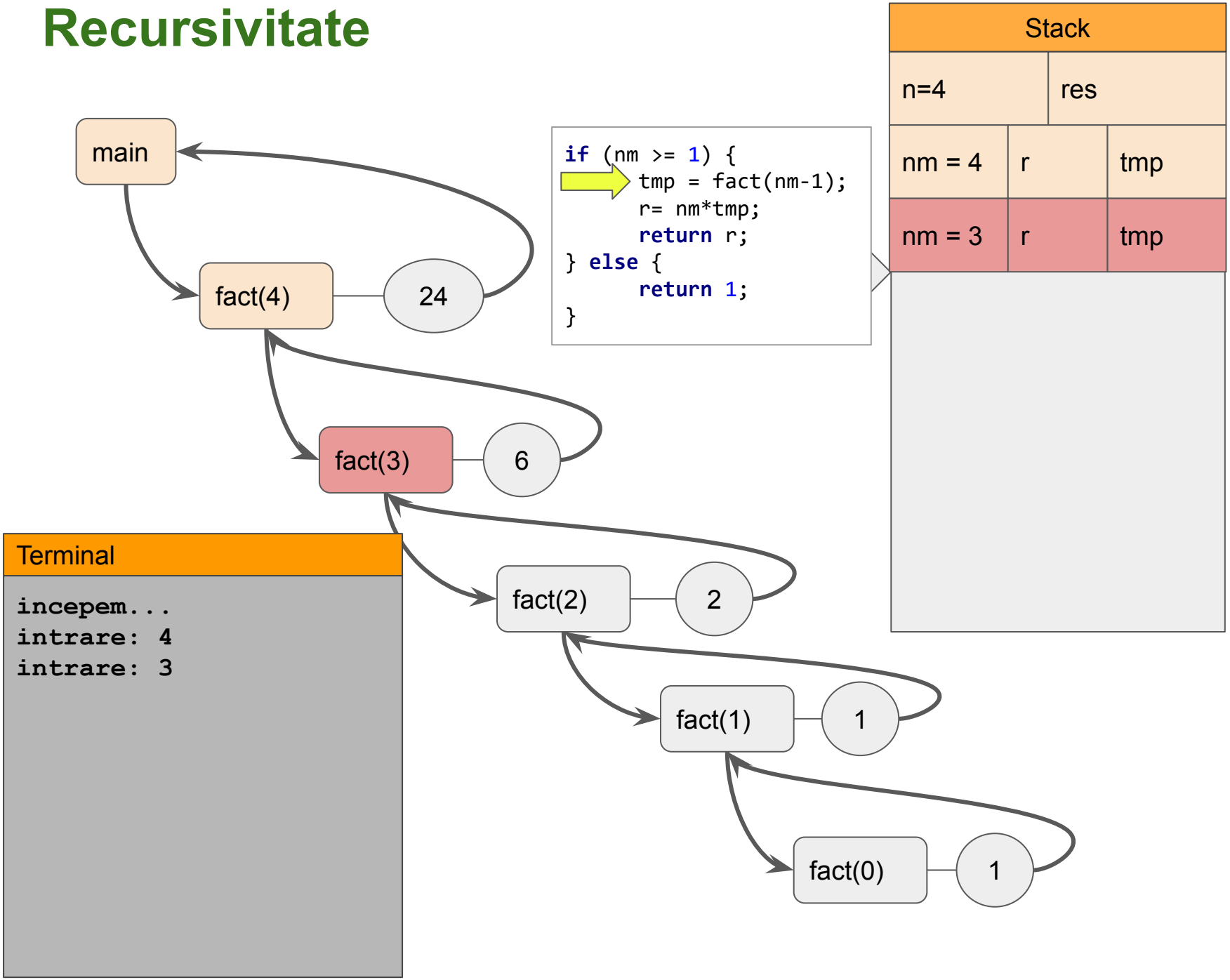
Recursivitate



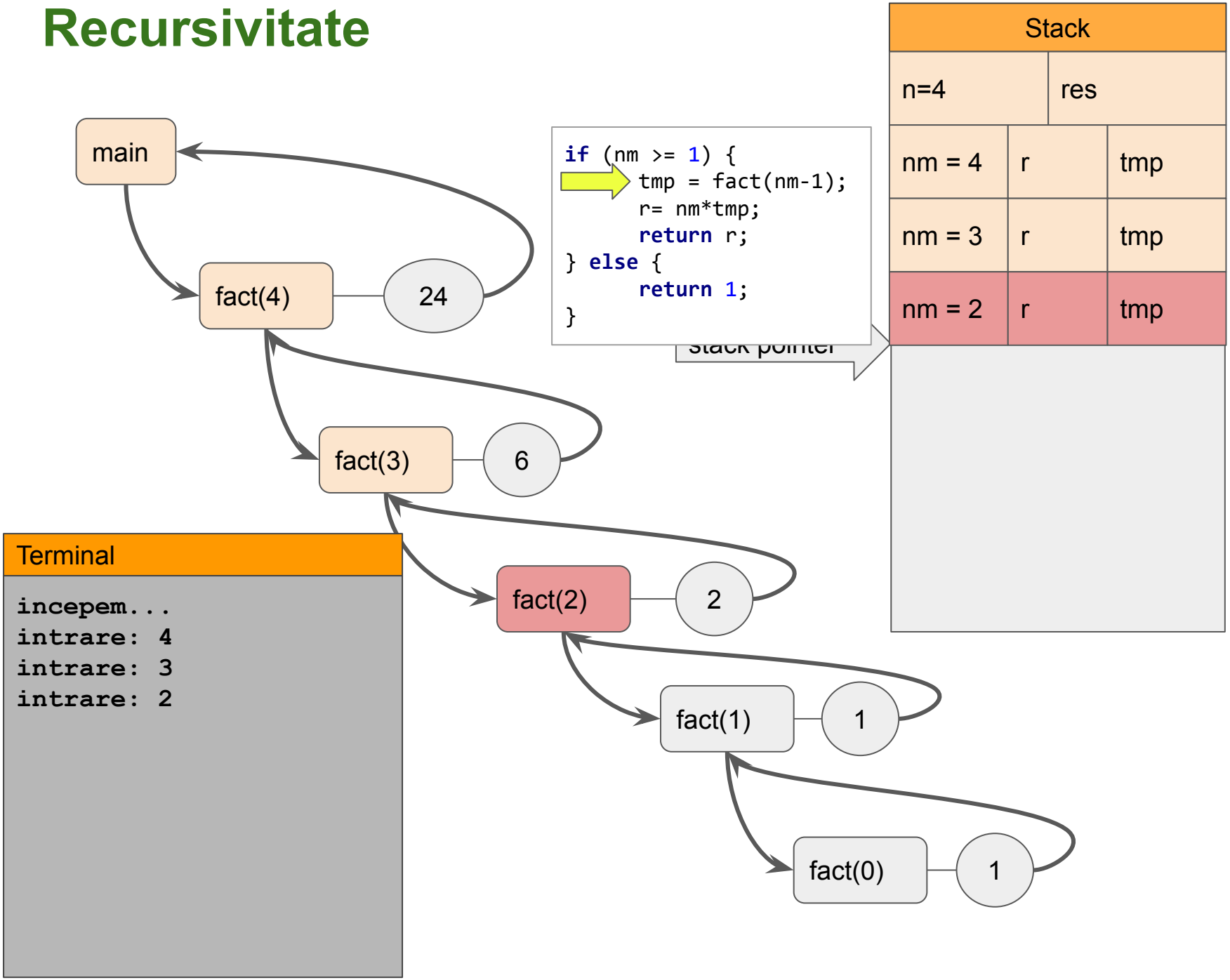
Recursivitate



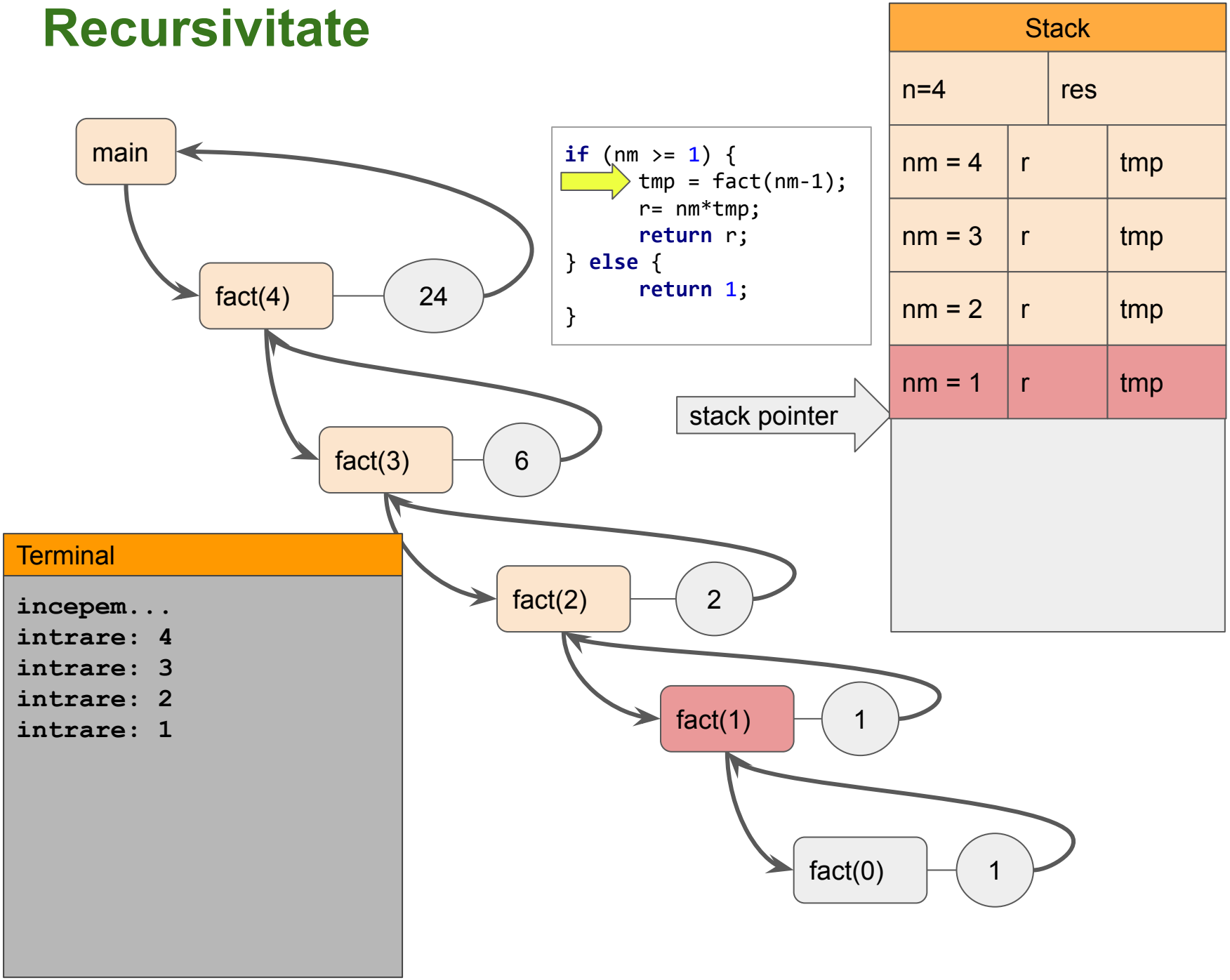
Recursivitate



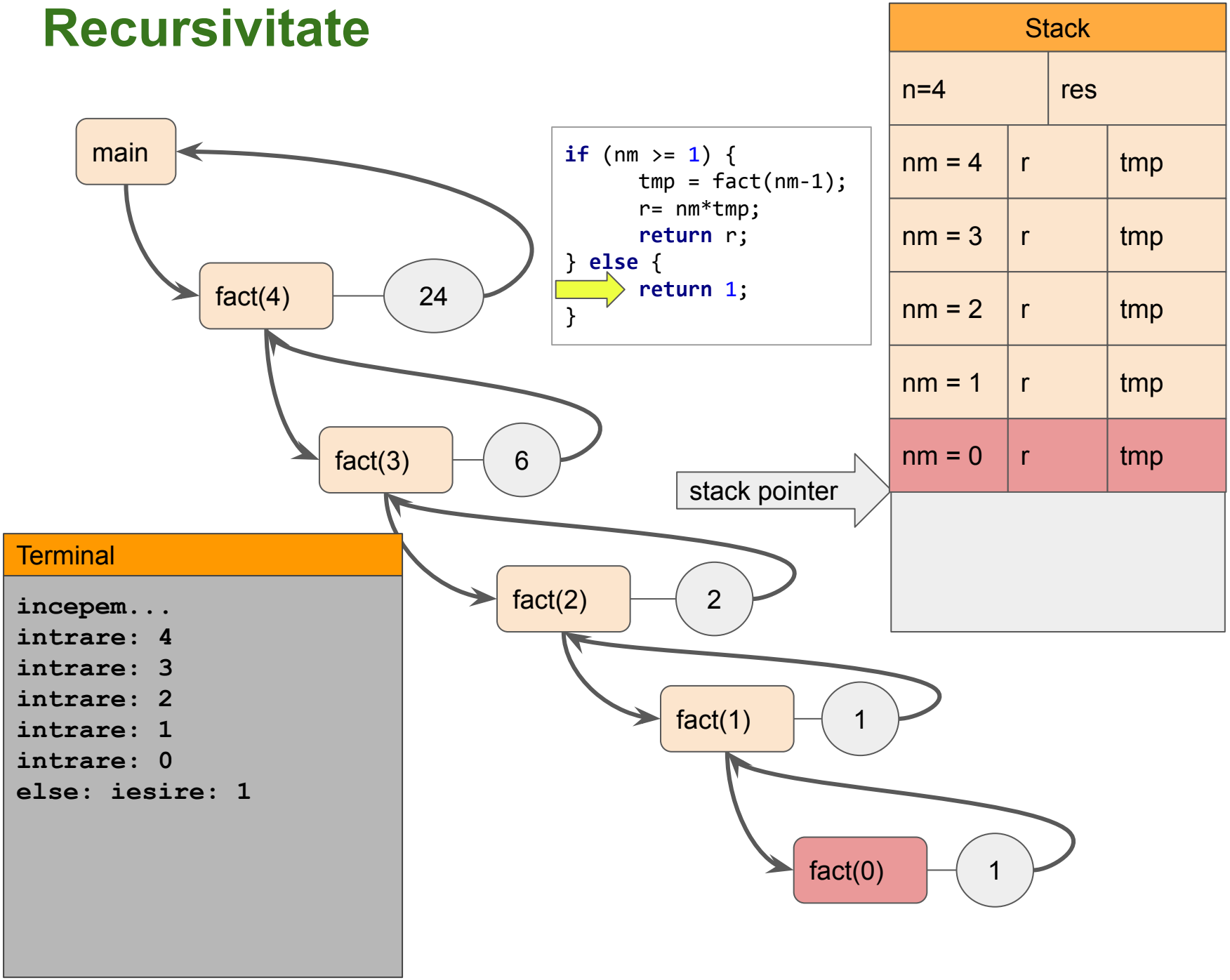
Recursivitate



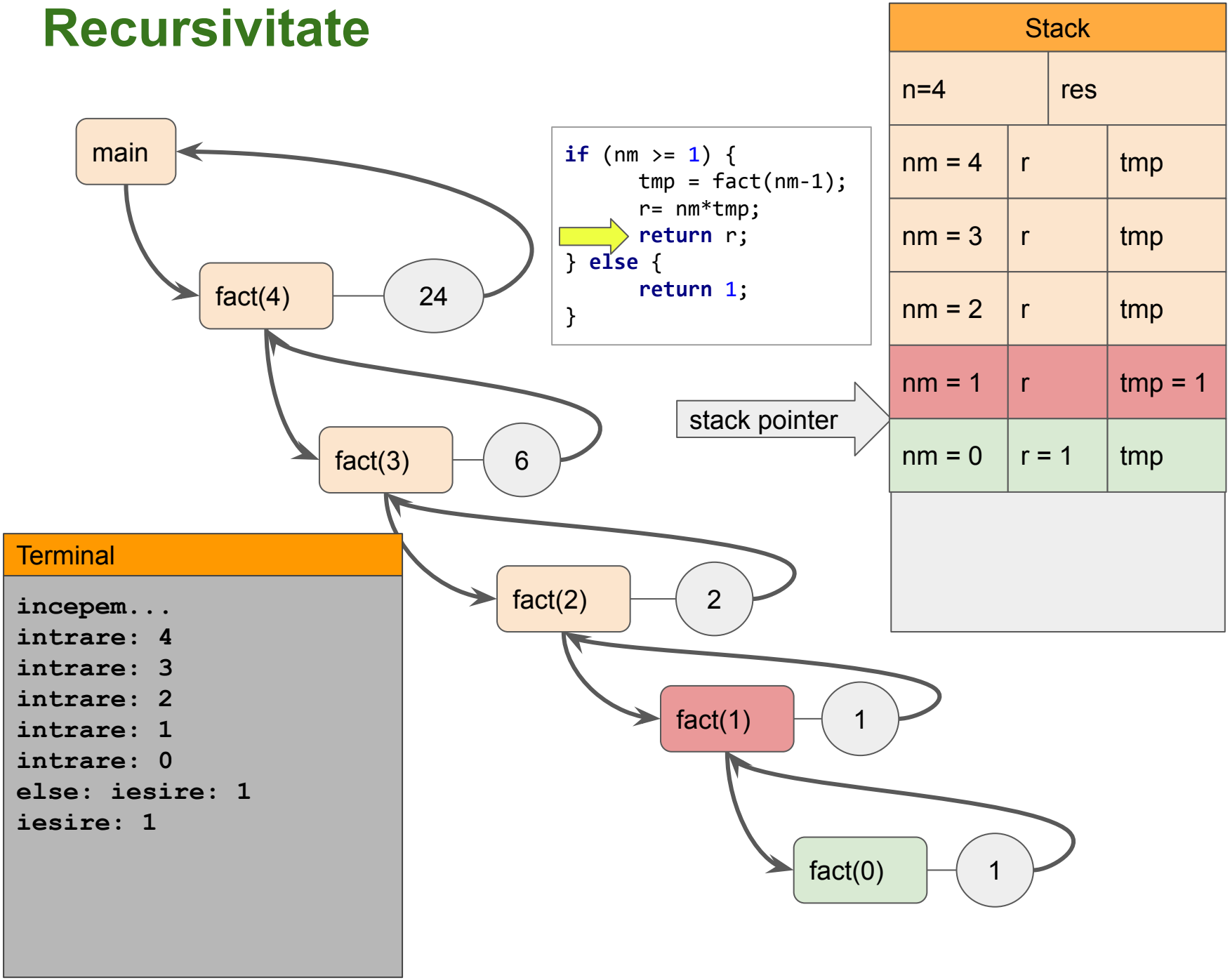
Recursivitate



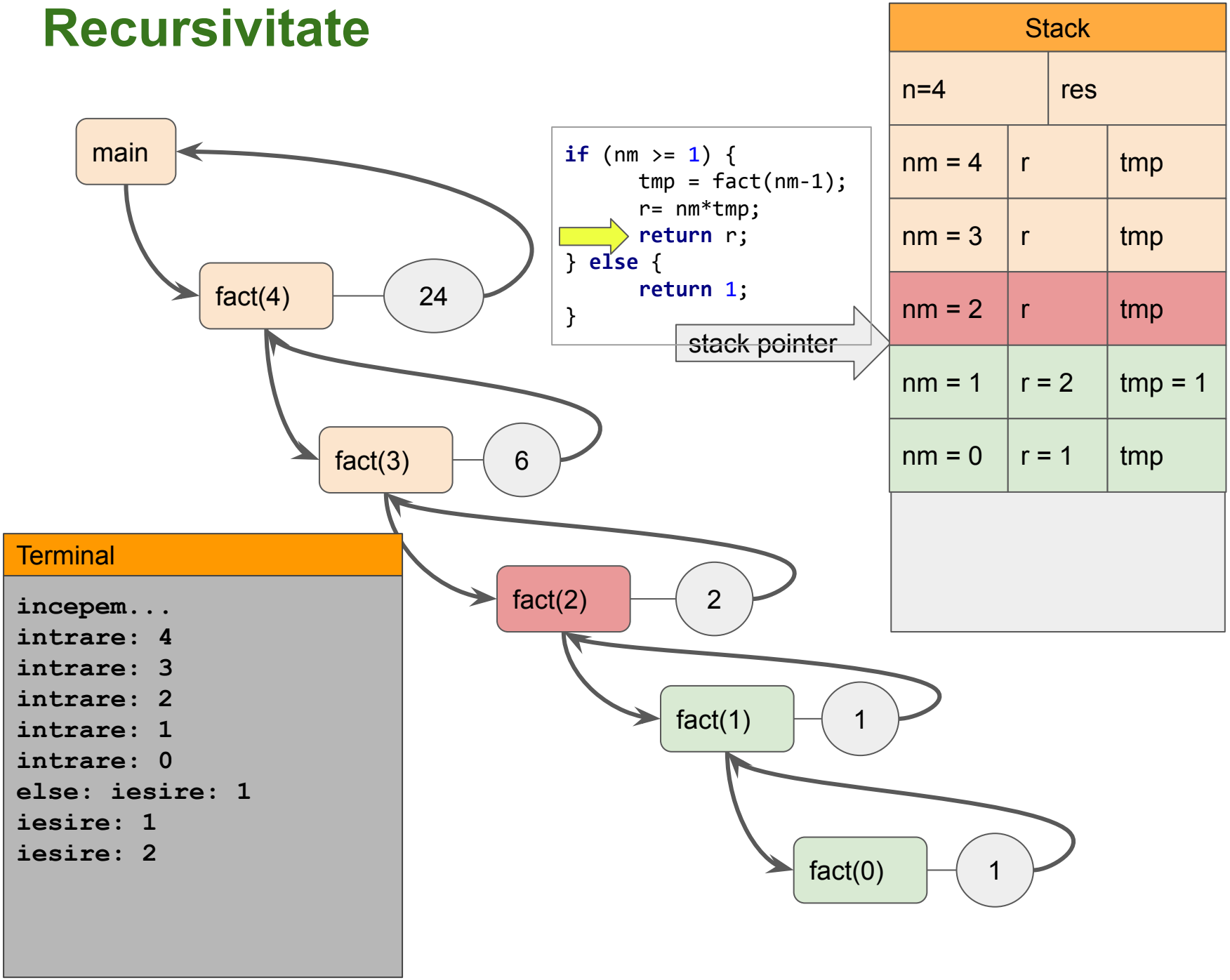
Recursivitate



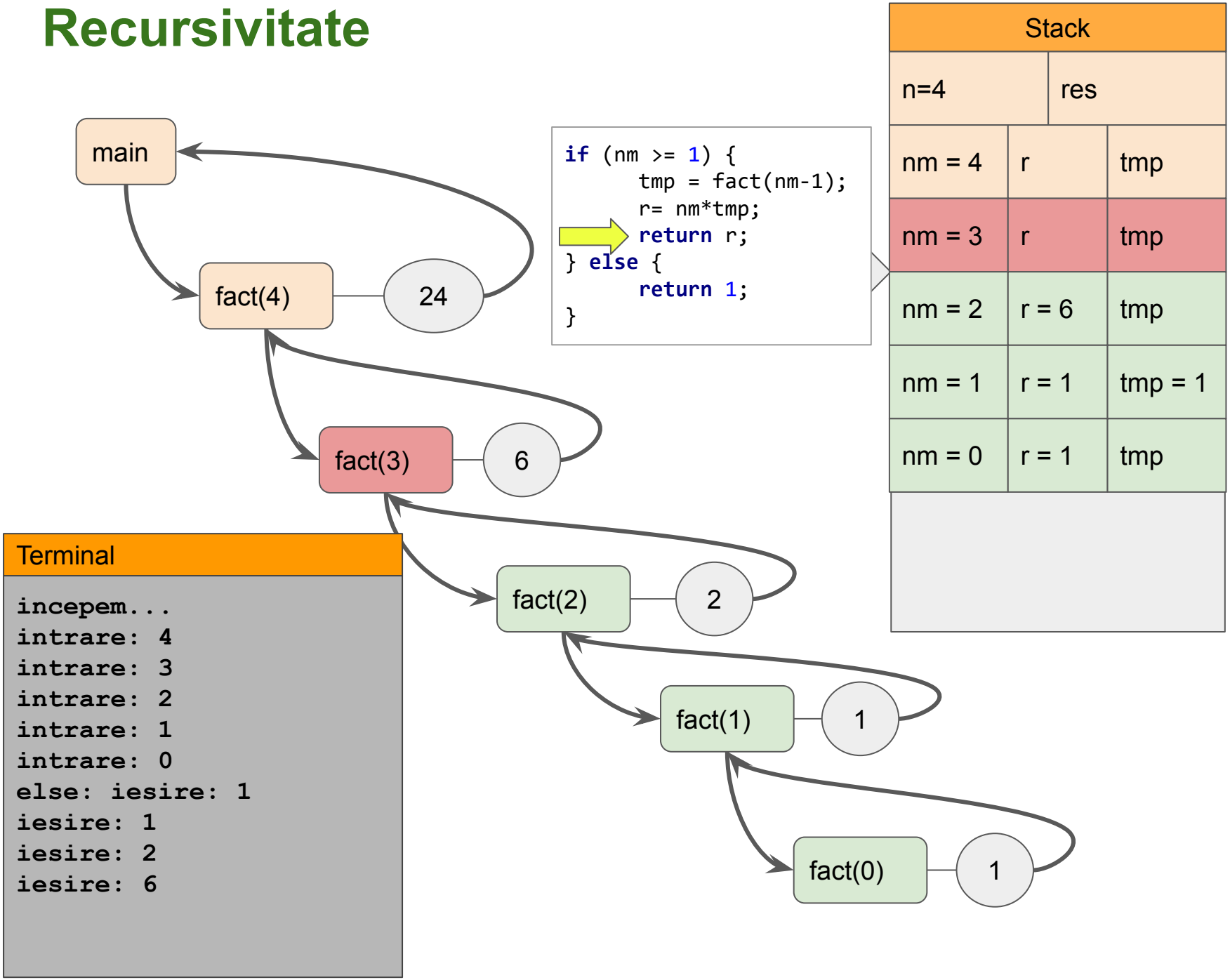
Recursivitate



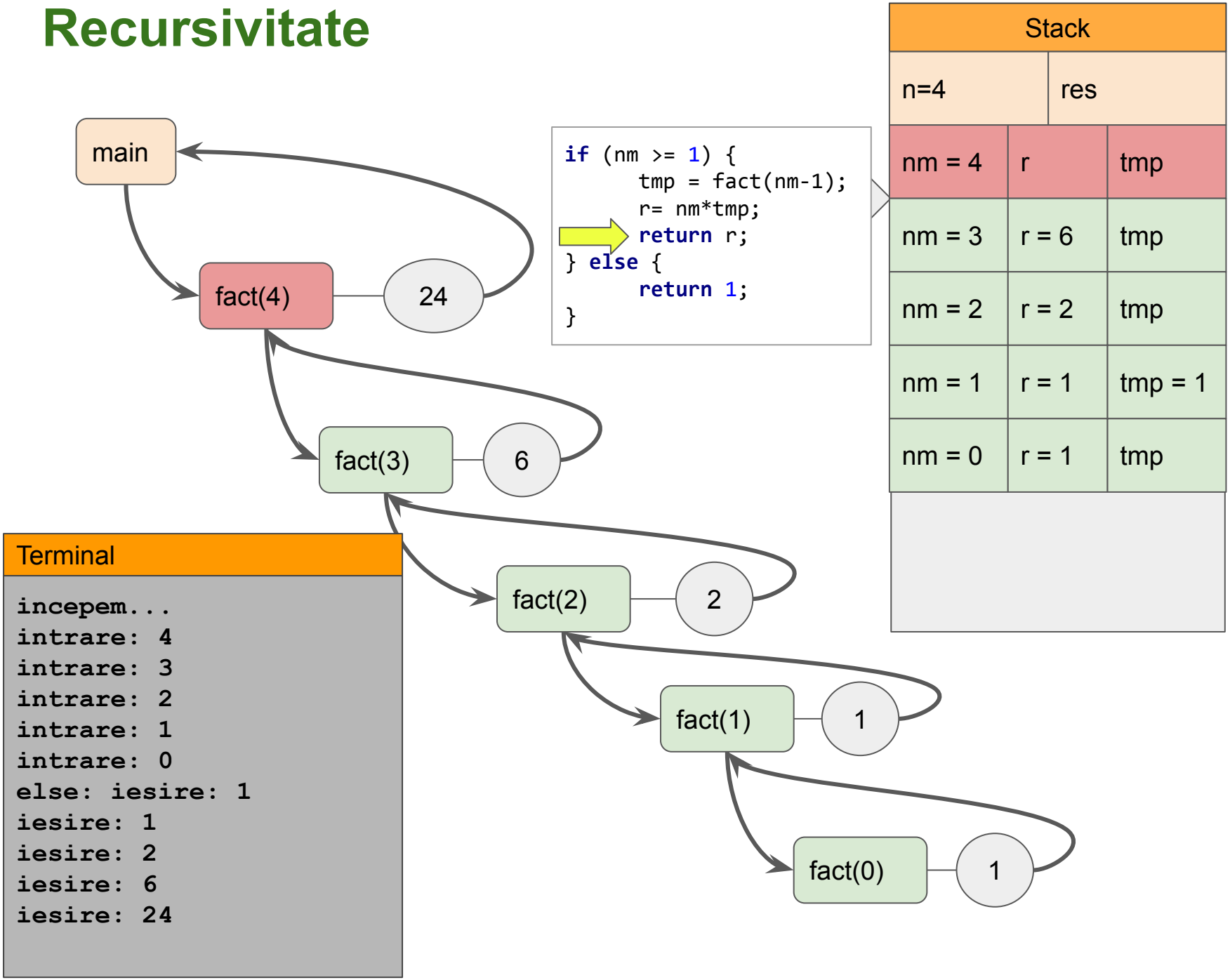
Recursivitate



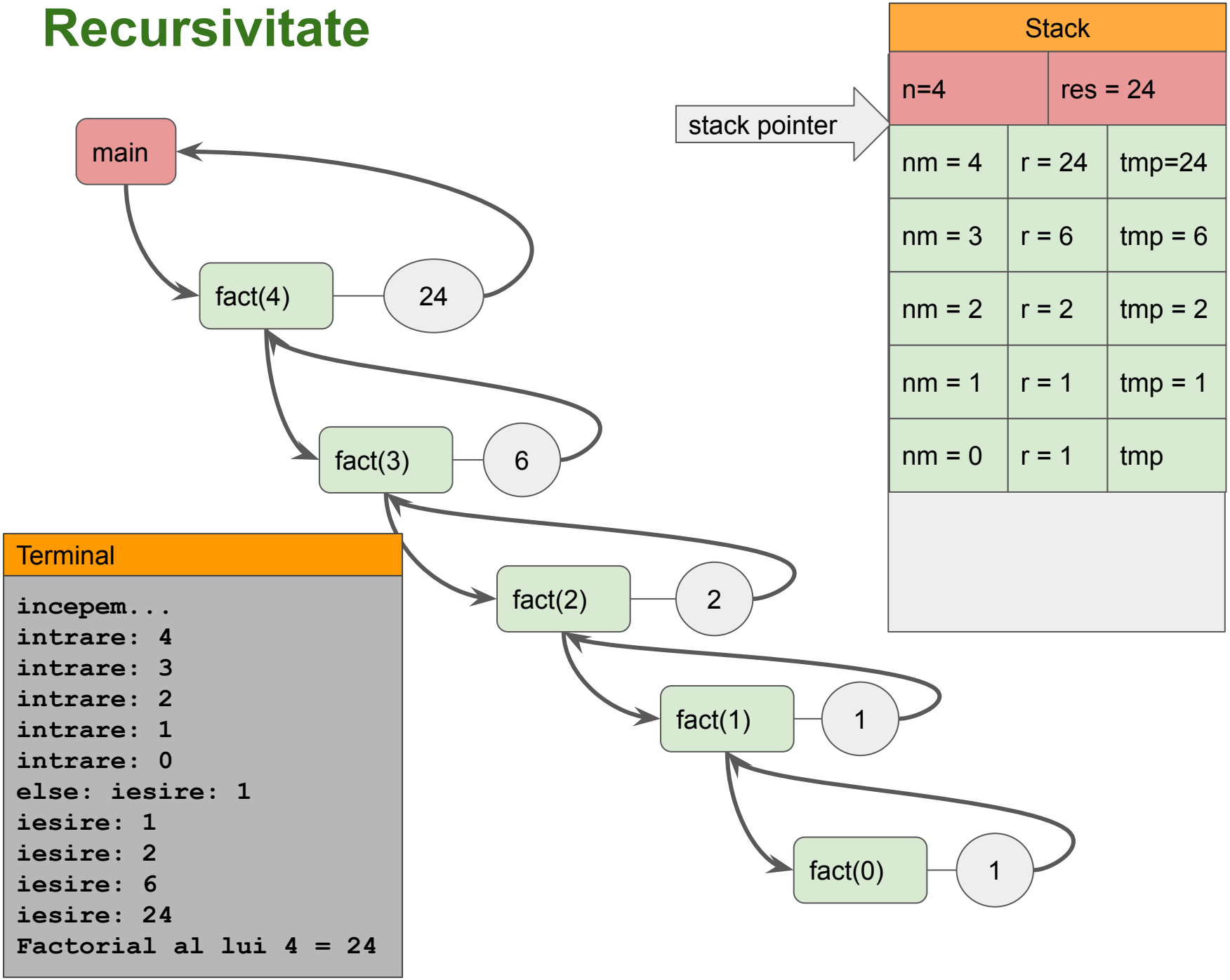
Recursivitate



Recursivitate



Recursivitate



Recursivitate

4! = 1 * 2 * 3 * 4 = 24

- Numărul care este trimis funcției recursive este 4. Cu toate acestea, datorită modului în care funcționează recursivitatea, înmulțirile încep de la 1.
- Programul a mers pe drumul 4 -> 3 ->2 -> 1 si abia apoi, după ce a terminat parcurgerea secvenței până la capăt, a început să se retragă, efectuând înmulțirile. Retragerea se face pe același drum datorită modului în care apelurile funcțiilor sunt dispuse succesiv în stack.

- Recursivitatea e mai lentă decât iterația. Pentru fiecare apel este nevoie de crearea unui stack frame, scriere/citire din memorie și registrele procesorului.
- Stack-ul poate fi insuficient pentru numărul de repetări necesare ale funcției
- Există pericolul ca anumite condiții să conducă către recursivitate infinită.
- Fiecare implementare a recursivității are o implementare iterativă echivalentă.
- Menținerea informațiilor de stare este dificilă

Suma recursivă a elementelor unei matrici

```c
#include <stdio.h>

int sum(int arr[], int start, int len);

int main()
{
   int arr[5] = {12,32,45,21,8};
   int sa = sum(arr, 0, 5);
   printf("Sum: %d\n", sa);
   return 0;
}

int sum(int arr[], int start, int len)
{
   printf ("Start: %d, Len: %d\n", start, len);
   if(start >= len) {
       printf("Rstop 0\n");
       return 0;
   } else {
       int ret = (arr[start] + sum(arr, start + 1, len));
       printf("Returning %d\n", ret);
       return ret;
   }
}
```

```
Start: 0, Len: 5
Start: 1, Len: 5
Start: 2, Len: 5
Start: 3, Len: 5
Start: 4, Len: 5
Start: 5, Len: 5
Rstop 0
Returning 8
Returning 29
Returning 74
Returning 106
Returning 118
Sum: 118
```

```
Sum: 118
```

https://onlinegdb.com/rkZ-D8f94

Utilizarea recursivității

- Căutare binară
- Alți algoritmi de tipul divide et impera
- Parcurgerea unei structuri arborescente
- Parcurgerea oricărei structuri de date repetitive a cărui lungime este necunoscută (dar care nu pune stack-ul în pericol).

Structuri

Obiectele reale sunt caracterizate de mai multe proprietăți, de cele mai multe ori exprimate prin tipuri de date diferite. Avem nevoie de o modalitate de a defini tipuri de date complexe care să ne permită descrierea acestora

Nume	Rubin
Nume științific	Corindon
Formulă chimică	Al2O3
Duritate	9
Indice de refracție	1.76
Fotografie	ruby.jpg
Culoare	rosu

Nume	Safir
Nume științific	Corindon
Formulă chimică	Al2O3
Duritate	9
Indice de refracție	1.76
Fotografie	safir.jpg
Culoare	albastru

Nume	Smarald
Nume științific	Beril
Formulă chimică	Be3Al2Si6O18
Duritate	8
Indice de refracție	1.59
Fotografie	smarald.jpg
Culoare	verde

Nume	Diamant
Nume științific	Diamant
Formulă chimică	C
Duritate	10
Indice de refracție	2,41
Fotografie	diamant.jpg
Culoare	incolor

gems_array.c

```c
#include <stdio.h>
#include <string.h>

int main()
{
   int indc;
   char *nume[4];
   char *snume[4];
   char *formula[4];
   int duritate[4];
   float refr[4];
   char *foto[4];
   char *clr[4];

   nume[0] = "Rubin"; nume[1] = "Safir"; nume[2] = "Smarald"; nume[3] = "Diamant";
   snume[0] = "Corindon"; snume[1] = "Corindon"; snume[2] = "Beril"; snume[3] = "Diamant";
   formula[0] = "Al2O3";formula[1] = "Al2O3";formula[2] = "Be3Al2Si6O18"; formula[3] = "C";
   duritate[0] = 9;duritate[1] = 9;duritate[2] = 8;duritate[3] = 10;
   refr[0] = 1.76;refr[1] = 1.76;refr[2] = 1.59;refr[3] = 2.41;
   foto[0] = "ruby.jpg"; foto[1] = "safir.jpg";foto[2] = "smarald.jpg";foto[3] = "diamant.jpg";
   clr[0] = "rosu";clr[1] = "albastru";clr[2] = "verde"; clr[3] = "incolor";

   puts ("Introduceti indicele (intre 0 si 3)\n");
   scanf("%d", &indc);
   printf("Intrarea cu numarul %d\n\n", indc);
   printf("Nume: %s\n",nume[indc]);
   printf("Nume stiintific: %s\n",snume[indc]);
   printf("Formula chimica: %s\n",formula[indc]);
   printf("Duritate: %d\n",duritate[indc]);
   printf("Indice de refr: %f\n",refr[indc]);
   printf("Fotografie: %s\n",foto[indc]);
   printf("Culoare: %s\n",clr[indc]);

   return 0;
}
```

Declarăm câte o matrice unidimensională pentru fiecare dintre proprietățile obiectelor. Fiecare matrice este declarată cu tipul necesar (char[], int, float) și dimensionată astfel încât să ne asigurăm că indicii corespund între ei.

https://onlinegdb.com/S16tBav54

```
Introduceti indicele (intre 0 si 3)
2
Intrarea cu numarul 2

Nume: Smarald
Nume stiintific: Beril
Formula chimica: Be3Al2(SiO3)6
Duritate: 8
Indice de refr: 1.590000
Fotografie: smarald.jpg
Culoare: verde
```

Structurile permit gruparea mai multor tipuri de date

struct_stone.c

```c
#include <stdio.h>
#include <string.h>

struct Stone
    {
        char nume[50];
        char snume[50];
        char formula[50];
        int duritate;
        float refr;
        char foto[30];
        char clr[30];
    };


int main()

{
   struct Stone rubin, safir, smarald, diamant;

   strcpy(rubin.nume, "Rubin");
   strcpy(rubin.snume, "Corindon");
   strcpy(rubin.formula, "Al2O3");

   printf ("Nume: %s\n", rubin.nume);
   printf ("Nume stiintific: %s\n", rubin.snume);
   printf ("Formula chimica: %s\n", rubin.formula);

}
```

Structurile se definesc de obicei în afara funcției main() pentru a avea un scop global.

Structurile sunt tipuri complexe de date, nu grupări de variabile. După definirea lor, trebuiesc declarate variabile cu noul tip.

Accesarea membrilor unei structuri (citire și scriere) se face cu ajutorul operatorului punct (.)

```
Nume: Rubin
Nume stiintific: Corindon
Formula chimica: Al2O3
```

https://onlinegdb.com/B1m2rpvc4

Declararea directă a structurilor

struct_stone_direct.c

```c
#include <stdio.h>
#include <string.h>

struct Stone
   {
       char nume[50];
       char snume[50];
       char formula[50];
       int duritate;
       float refr;
       char foto[30];
       char clr[30];
   };

int main()

{
   struct Stone rubin = {"Rubin", "Corindon", "Al2O3", 9, 1.76, "ruby.jpg", "rosu"};
   struct Stone safir = {"Safir", "Corindon", "Al2O3", 9, 1.76, "safir.jpg", "albastru"};
   struct Stone smarald = {"Smarald", "Beril", "Be3Al2Si6O18", 8, 1.59, "smarald.jpg", "verde"};
   struct Stone diamant = {"Diamant", "Diamant", "C", 10, 2.41, "diamant.jpg", "incolor"};

   puts ("Rubin\n");
   printf ("Nume: %s\n", rubin.nume);
   printf ("Nume stiintific: %s\n", rubin.snume);
   printf ("Formula chimica: %s\n", rubin.formula);
   puts ("\nSmarald\n");
   printf ("Nume: %s\n", smarald.nume);
   printf ("Nume stiintific: %s\n", smarald.snume);
   printf ("Formula chimica: %s\n", smarald.formula);
}
```

```
Rubin

Nume: Rubin
Nume stiintific: Corindon
Formula chimica: Al2O3

Smarald

Nume: Smarald
Nume stiintific: Beril
Formula chimica: Be3Al2Si6O18
```

https://onlinegdb.com/BJKQH5WOu

Șirurile din structuri se pot defini și ca pointeri

struct_stone_pointers.c

```c
#include <stdio.h>
#include <string.h>

struct Stone
    {
        char *nume;
        char *snume;
        char *formula;
        int duritate;
        float refr;
        char *foto;
        char *clr;
    };


int main()

{
    struct Stone rubin, safir, smarald, diamant;

    rubin.nume =  "Rubin";
    rubin.snume = "Corindon";
    rubin.formula = "Al2O3";

    printf ("Nume: %s\n", rubin.nume);
    printf ("Nume stiintific: %s\n", rubin.snume);
    printf ("Formula chimica: %s\n", rubin.formula);
}
```

https://onlinegdb.com/SJwUDTv5E

Structurile se definesc de obicei în afara funcției main() pentru a avea un scop global.

Structurile sunt tipuri complexe de date, nu grupări de variabile. După definirea lor, trebuiesc declarate variabile cu noul tip.

Accesarea membrilor unei structuri (citire și scriere) se face cu ajutorul operatorului punct (.)

Functionează fără malloc pentru că sirurile sunt declarate în momentul compilării.

```
Nume: Rubin
Nume stiintific: Corindon
Formula chimica: Al2O3
```

Structurile pot fi accesate și cu ajutorul pointerilor

pointers_to_struct.c

```c
#include <stdio.h>
#include <string.h>

struct Stone
   {
       char *nume;
       char *snume;
       char *formula;
       int duritate;
       float refr;
       char *foto;
       char *clr;
   };

int main()

{
   struct Stone *prubin, *psafir, *psmarald, *pdiamant, rubin, safir, smarald, diamant;

   prubin = &rubin; psafir = &safir; pdiamant = &diamant; psmarald = &smarald;

   prubin->nume =  "Rubin";
   prubin->snume = "Corindon";
   prubin->formula = "Al2O3";

   printf ("Nume: %s\n", prubin->nume);
   printf ("Nume stiintific: %s\n", prubin->snume);
   printf ("Formula chimica: %s\n", prubin->formula);
}
```

pointerii se definesc cu același tip de date (strucutra definită mai sus)

Pointerilor li se atribuie adresa fiecărei variabile (ca de obicei)

Apelarea membrului unui variabile de tip structură se face prin operatorul (->) care dereferențiază automat.

```
Nume: Rubin
Nume stiintific: Corindon
Formula chimica: Al2O3
```

https://onlinegdb.com/ryG2wTvc4

Dacă pointerii se dereferențiază, se revine la operatorul (.)

pointers_to_struct_deref.c

```c
#include <stdio.h>
#include <string.h>

struct Stone
   {
       char *nume;
       char *snume;
       char *formula;
       int duritate;
       float refr;
       char *foto;
       char *clr;
   };


int main()

{
   struct Stone *prubin, *psafir, *psmarald, *pdiamant, rubin, safir, smarald, diamant;

   prubin = &rubin; psafir = &safir; pdiamant = &diamant; psmarald = &smarald;

   (*prubin).nume =  "Rubin";
   (*prubin).snume = "Corindon";
   (*prubin).formula = "Al2O3";

   printf ("Nume: %s\n", (*prubin).nume);
   printf ("Nume stiintific: %s\n", (*prubin).snume);
   printf ("Formula chimica: %s\n", (*prubin).formula);
}
```

pointerii se definesc cu același tip de date (strucutra definită mai sus)

Pointerilor li se atribuie adresa fiecărei variabile (ca de obicei)

(*prubin) = rubin

```
Nume: Rubin
Nume stiintific: Corindon
Formula chimica: Al2O3
```

https://onlinegdb.com/HJJyOaP54

Structurile se pot grupa în șiruri

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

struct Stone
   {
       char *nume;
       char *snume;
       char *formula;
       int duritate;
       float refr;
       char *foto;
       char *clr;
   };


int main()

{
   struct Stone stones[4];

   stones[0].nume =  "Rubin";
   stones[0].snume = "Corindon";
   stones[0].formula = "Al2O3";

   printf ("Nume: %s\n", stones[0].nume);
   printf ("Nume stiintific: %s\n", stones[0].snume);
   printf ("Formula chimica: %s\n", stones[0].formula);
}
```

```
Nume: Rubin
Nume stiintific: Corindon
Formula chimica: Al2O3
```

arrays_of_structs.c

Șir de 4 variabile de tip Stone

Variabilele se adresează cu ajutorul index-ului șirului

https://onlinegdb.com/SkT-_TDcV

Typedef

Typedef este un operator care permite definirea unor alias-uri pentru tipurile de variabile.

typedefs.c

```c
#include <stdio.h>

typedef int asprete;
typedef char salau;
typedef float babusca;
typedef short int platica;

int main()
{
   asprete lng = 12213;
   salau cr = 'd';
   salau decl[] = "Nu stiu nimic";
   babusca srs = 12.6576;
   platica mm = 12;

   printf("lng: %d\n", lng);
   printf("cr: %c\n", cr);
   printf("decl: %s\n", decl);
   printf("srs: %f\n", srs);
   printf("mm: %d\n", mm);
   return 0;
}
```

https://onlinegdb.com/SJLduaP5N

```
lng: 12213
cr: d
decl: Nu stiu nimic
srs: 12.657600
mm: 12
```

typedef este extrem de utilizat, de pildă, în sursa nucleului linux (4.9.51) sunt 16338 de apariții

```c
typedef unsigned char bits8;
typedef signed char sbits8;
typedef unsigned short int bits16;
typedef signed short int sbits16;
typedef unsigned int bits32;
typedef signed int sbits32;
typedef unsigned long long int uint64;
typedef signed long long int int64;
```

Se folosește foarte frecvent pentru scurtarea numelui tipurilor de variabile

Typedef în definirea structurilor

typedef_struct.c

```c
#include <stdio.h>
#include <string.h>

typedef struct
   {
       char *nume;
       char *snume;
       char *formula;
       int duritate;
       float refr;
       char *foto;
       char *clr;
   } bolovan;


int main()

{
   bolovan stones[4];

   stones[0].nume =  "Rubin";
   stones[0].snume = "Corindon";
   stones[0].formula = "Al2O3";

   printf ("Nume: %s\n", stones[0].nume);
   printf ("Nume stiintific: %s\n", stones[0].snume);
   printf ("Formula chimica: %s\n", stones[0].formula);
}
```

utilizarea typedef împreună cu struct permite crearea aliasului bolovan pentru structura. Denumirea structurii nu mai este necesara.

Declararea variabilelor se face direct cu aliasul definit.

https://onlinegdb.com/Hk9w56DcE

Dimensiunea unei structuri

struct_dim_24.c

```c
#include <stdio.h>

typedef struct  {
   int a;  // 4 octeti
   int c;  //4 octeti
   double b;   // 8 octeti
   long long int d; // 8 octeti
} VARS;

int main() {
   printf("dim v1 %ld\n", sizeof(VARS) );
   return 0;
}
```

https://onlinegdb.com/B-zbTVcIT

```
dim v1 24
```

struct_dim_32.c

```c
#include <stdio.h>

typedef struct  {
   int a;  // 4 octeti
   double b;   // 8 octeti
   int c;  //4 octeti
   long long int d; // 8 octeti
} VARS;

int main() {
   printf("dim v1 %ld\n", sizeof(VARS) );
   return 0;
}
```

https://onlinegdb.com/lZpc9oL0h

```
dim v1 32
```

```c
int a;  // 4 octeti
int c;  //4 octeti
double b;   // 8 octeti
long long int d; // 8 octeti
```

4 + 4 + 8 + 8 = 24

Sigura diferență între cele două declarații ale structurii este faptul că au fost inversate variabilele c cu b.

4 + 8 + 4 + 8 = 32 ??

```c
int a;  // 4 octeti
double b;   // 8 octeti
int c;  //4 octeti
long long int d; // 8 octeti
```

Dimensiunea unei structuri

```
int a;   // 4 octeti
int c;   //4 octeti
double b;    // 8 octeti
long long int d; // 8 octeti
```

Structurile sunt stocate în memorie în zone continue, membrii lor fiind așezați unui după celălalt, în ordinea în care au fost declarați.

4 + 4 + 8 + 8 = 24

4 + 8 + 4 + 8 = 32 ??

```
int a;   // 4 octeti
double b;    // 8 octeti
int c;   //4 octeti
long long int d; // 8 octeti
```

Dimensiunea unei structuri (întotdeauna cu sizeof!)

Chiar dacă memoria este adresabilă la nivel de octet, procesorul citește atâția octeți câți încap într-un registru (8 octeți în cazul unui procesor pe 64 de biți).

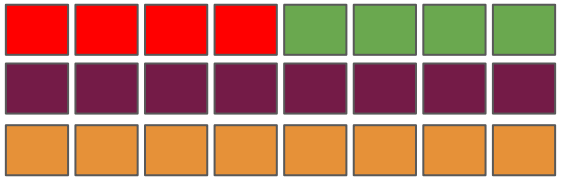
Compilatoarele aliniază variabilele astfel încât sa înceapă toate la cel mai din stânga octet. O variabilă nu poate fi citită în două citiri succesive. Din acest motiv, în funcție de ordinea declarării variabilelor, pot rămâne spații libere.

Pentru a încărca toată structura, în primul caz este nevoie de 24 de biți iar în al doilea de 32

Din acest motiv C nu permite comparația structurilor.

```
int a;   // 4 octeti
int c;   //4 octeti
double b;    // 8 octeti
long long int d; // 8 octeti
```

4 + 4 + 8 + 8 = 24



24

```
int a;   // 4 octeti
double b;    // 8 octeti
int c;   //4 octeti
long long int d; // 8 octeti
```

4 + 8 + 4 + 8 = 32

32

Copierea unei structuri în altă structură cu aceeași definiție

Există trei metode pentru a copia o structură definită și inițializată într-o altă structură cu aceeași definiție:

- Copierea prin atribuire.
- Copierea manuală, membru cu membru.
- Copierea memoriei ocupate de structură, octet cu octet, folosind funcția memcpy.

Atribuire

struct_copy_attr.c

```c
#include <stdio.h>

typedef struct {
   char *start;
   char *end;
} mutare;

int main() {

   mutare m1, m2;
   m1.start = "E2";
   m1.end = "E4";
   printf ("M1.start = %s\n", m1.start);
   printf ("M1.end = %s\n", m1.end);
   m2 = m1;
   printf ("M2.start = %s\n", m2.start);
   printf ("M2.end = %s\n", m2.end);
}
```

https://onlinegdb.com/r1Sjq6wcE

```
M1.start = E2
M1.end = E4
M2.start = E2
M2.end = E4
```

membru cu membru

struct_copy_man.c

```c
#include <stdio.h>

typedef struct mutare {
   char *start;
   char *end;
} mutare;

int main() {
   mutare m1, m2;
   m1.start = "E2";
   m1.end = "E4";
   printf ("M1.start = %s\n", m1.start);
   printf ("M1.end = %s\n", m1.end);
   m2.start = m1.start;
   m2.end = m1.end;
   printf ("M2.start = %s\n", m2.start);
   printf ("M2.end = %s\n", m2.end);
}
```

https://onlinegdb.com/ryCpcpvq4

```
M1.start = E2
M1.end = E4
M2.start = E2
M2.end = E4
```

Copierea structurilor cu memcpy

struct_memcpy.c

```c
#include <stdio.h>
#include <string.h>

typedef struct {
   char *start;
   char *end;
} mutare;


int main() {

   mutare m1, m2;
   m1.start = "E2";
   m1.end = "E4";
   printf ("M1.start = %s\n", m1.start);
   printf ("M1.end = %s\n", m1.end);

   memcpy (&m2, &m1, sizeof(mutare));

   printf ("M2.start = %s\n", m2.start);
   printf ("M2.end = %s\n", m2.end);

}
```

https://onlinegdb.com/SJYlipD94

void *memcpy(void *dest, const void *src, size_t n)

memcpy copiaza numărul de octeți din argumentul n începând de la adresa pointerului src, în spațiul echivalent de memorie care începe de la adresa pointerului dest.

Atenție la pointeri la copierea structurilor

struct_shallow_copy.c

```c
#include <stdio.h>
#include <string.h>

typedef struct {
   int *p1;
   int *p2;
} pointeri;

int main() {
   int fvar = 12;
   int *pfvar = &fvar;
   int svar = 123;
   int *psvar = &svar;

   pointeri POI1, POI2;

   POI1.p1 = pfvar;
   POI1.p2 = psvar;
   printf ("POI1.p1 = %d\n", *POI1.p1);
   printf ("POI1.p2 = %d\n", *POI1.p2);
   printf ("Ad POI1.p1 = %p\n", POI1.p1);
   printf ("Ad POI1.p2 = %p\n", POI1.p2);

   POI2 = POI1;
   fvar = 1921;

   printf ("POI1.p1 = %d\n", *POI1.p1);
   printf ("POI2.p1 = %d\n", *POI2.p1);
   printf ("POI2.p2 = %d\n", *POI2.p2);
   printf ("Ad POI2.p1 = %p\n", POI2.p1);
   printf ("Ad POI2.p2 = %p\n", POI2.p2);
}
```

https://onlinegdb.com/Sk7Xs6v5N

Shallow copy - se copiază pointerii dintr-o structură în alta iar la final ambele structuri vor trimite către aceleași date. Modificarea uneia dintre date (sau chiar a variabilei inițiale) determină modificarea datelor în ambele structuri.

Toate metodele de copiere prezentate fac shallow copy.

Deep copy - se aloca memorie, se dubleaza toate datele la care structura are acces, se definesc noi pointeri către noile zone de memorie.

Funcțiile pentru deep copy trebuie programate.

```
POI1.p1 = 12
POI1.p2 = 123
Ad POI1.p1 = 0x7fffb93d7f38
Ad POI1.p2 = 0x7fffb93d7f3c
POI1.p1 = 1921
POI2.p1 = 1921
POI2.p2 = 123
Ad POI2.p1 = 0x7fffb93d7f38
Ad POI2.p2 = 0x7fffb93d7f3c
```

Deep copy

struct_deep_copy.c

```c
#include <stdio.h>
#include <string.h>

typedef struct {
   int *p1;
   int *p2;
} pointeri;

int main() {
   int fvar = 12;
   int *pfvar = &fvar;
   int svar = 123;
   int *psvar = &svar;

   pointeri POI1, POI2;

   POI1.p1 = pfvar;
   POI1.p2 = psvar;
   printf ("POI1.p1 = %d\n", *POI1.p1);
   printf ("POI1.p2 = %d\n", *POI1.p2);
   printf ("Ad POI1.p1 = %p\n", POI1.p1);
   printf ("Ad POI1.p2 = %p\n", POI1.p2);

   POI2 = POI1;
   fvar = 1921;

   printf ("POI1.p1 = %d\n", *POI1.p1);
   printf ("POI2.p1 = %d\n", *POI2.p1);
   printf ("POI2.p2 = %d\n", *POI2.p2);
   printf ("Ad POI2.p1 = %p\n", POI2.p1);
   printf ("Ad POI2.p2 = %p\n", POI2.p2);
}
```

În funcție de complexitatea structurilor (care poate fi foarte mare) funcțiile de deep copy pot deveni extrem de complicate.

1. Se alocă un spațiu corespunzător de memorie pentru p1
2. Se alocă un spațiu corespunzător de memorie pentru p2
3. Se atribuie valoarea dereferentiată a POI1.p1 adresei către care trimite POI2.p1
4. Se atribuie valoarea dereferentiată a POI1.p2 adresei către care trimite POI2.p2

```c
POI2.p1 = (malloc(sizeof(int)));
POI2.p2 = (malloc(sizeof(int)));
*POI2.p1 = *POI1.p1;
*POI2.p2 = *POI1.p2;
```

În exemplul dat, fiind vorba de numere întregi, se putea evita utilizarea malloc, este dat ca exemplu doar.

Structuri imbricate

Structurile reprezintă o colecție formată din mai multe tipuri de date alăturate. Printre acestea, se pot afla și alte structuri.

```c
typedef struct {
   char *nume;
   char *simbol;
   int nrat;
} element;

typedef struct {
    char *nume;
    char *snume;
    int nel;
    element chimism[10];
} bolovan;
```

struct_in_struct.c

```c
element eal = {"Aluminiu", "Al", 13};
element eo = {"Oxigen", "O", 8};
element esi = {"Siliciu", "Si", 14};
element ebe = {"Beriliu", "Be", 4};
element ec = {"Carbon", "C", 6};
```

```c
stones[0].nume =  "Rubin";
stones[0].snume = "Corindon";
stones[0].nel = 2;
stones[0].chimism[0] = eal;
stones[0].chimism[1] = eo;

stones[1].nume =  "Smarald";
stones[1].snume = "Beril";
stones[1].nel = 4;
stones[1].chimism[0] = ebe;
stones[1].chimism[1] = eal;
stones[1].chimism[2] = esi;
stones[1].chimism[3] = eo;
```

O structură nu poate avea un membru care să aparțină tipului său. Poate avea un pointer de tipul său.

```c
printf ("Nume: %s\n", stones[0].nume);
for (int i = 0; i < stones[0].nel; i++) {
   printf ("--->%s Z: %d\n", stones[0].chimism[i].nume, stones[1].chimism[i].nrat);
}

printf ("Nume: %s\n", stones[1].nume);
for (int i = 0; i < stones[1].nel; i++) {
   printf ("--->%s Z: %d\n", stones[1].chimism[i].nume, stones[1].chimism[i].nrat);
}
```

https://onlinegdb.com/Sy6Osav94

Declarații explicite

Structurile reprezintă o colecție formată din mai multe tipuri de date alăturate. Printre acestea, se pot afla și alte structuri.

```
typedef struct {
   char *nume;
   char *simbol;
   int nrat;
} element;

typedef struct {
    char *nume;
    char *snume;
    int nel;
    element chimism[10];
} bolovan;
```

```
element eal = {"Aluminiu", "Al", 13};
element eo = {"Oxigen", "O", 8};
element esi = {"Siliciu", "Si", 14};
element ebe = {"Beriliu", "Be", 4};
element ec = {"Carbon", "C", 6};
```

struct_in_struct.c

Dimensiunea șirului chimism trebuie declarată, șirurile flexibile nu pot fi declarate în structuri care apar ca membrii altor structuri. În structuri primare se pot folosi dar cu condiția să fie ultimul element al șirului.

```
bolovan stones[] = {
                      {"Rubin", "Corindon", 2, {eal, eo}},
                      {"Smarald", "Beril", 4, {ebe, eal,esi,eo}}
                    };
```

```
printf ("Nume: %s\n", stones[0].nume);
for (int i = 0; i < stones[0].nel; i++) {
   printf ("--->%s Z: %d\n", stones[0].chimism[i].nume, stones[1].chimism[i].nrat);
}

printf ("Nume: %s\n", stones[1].nume);
for (int i = 0; i < stones[1].nel; i++) {
   printf ("--->%s Z: %d\n", stones[1].chimism[i].nume, stones[1].chimism[i].nrat);
}
```

Trimiterea structurilor către funcții

Structurile pot fi utilizate ca argumente în funcții. Structurile pot fi returnate de funcții.

struct_functions.c

```c
#include <stdio.h>
#include <string.h>

void printstruct(bolovan piatra);

int main()
{

   element eal = {"Aluminiu", "Al", 13};
   element eo = {"Oxigen", "O", 8};
   element esi = {"Siliciu", "Si", 14};
   element ebe = {"Beriliu", "Be", 4};
   element ec = {"Carbon", "C", 6};
   bolovan rubin = {"Rubin", "Corindon", 2, {eal, eo}};
   bolovan smarald = {"Smarald", "Beril", 4, {ebe, eal,esi,eo}};

   printstruct(smarald);
}

void printstruct (bolovan piatra) {
   printf ("Nume: %s\n", piatra.nume);
   printf ("Nume stiintific: %s\n", piatra.snume);
   for (int i = 0; i < piatra.nel; i++) {
       printf ("--->%s Z: %d\n", piatra.chimism[i].nume, piatra.chimism[i].nrat);
   }
}
```

```c
typedef struct {
   char *nume;
   char *simbol;
   int nrat;
} element;

typedef struct {
    char *nume;
    char *snume;
    int nel;
    element chimism[10];
} bolovan;
```

```
Nume: Smarald
Nume stiintific:
Beril
--->Beriliu Z: 4
--->Aluminiu Z: 13
--->Siliciu Z: 14
--->Oxigen Z: 8
```

https://onlinegdb.com/Sy9-T6P5E

Trimiterea structurilor către funcții - by value

Acest mod de trimitere a structurilor este “by value” - structura este copiata la intrarea în funcție iar la returnarea în funcția main, structura copiată este ștearsă. Operațiile din funcția changestruct nu afecteaza structura din main.

struct_changestruct.c

```c
#include <stdio.h>
#include <string.h>

void changestruct(bolovan piatra);

int main()
{
   element eal = { "Aluminiu", "Al", 13};
   element eo = { "Oxigen", "O", 8};
   element esi = { "Siliciu", "Si", 14};
   element ebe = { "Beriliu", "Be", 4};
   element ec = { "Carbon", "C", 6};
   bolovan smarald = { "Smarald", "Beril", 4, {ebe, eal,esi,eo}};
   printf ("Nume: %s\n", smarald.nume);
   changestruct(smarald);
   printf ("Nume: %s\n", smarald.nume);
}

void changestruct (bolovan piatra) {
   piatra.nume = "Diamant";
   printf ("Nume in changestruct: %s\n", piatra.nume);
}
```

```c
typedef struct {
   char *nume;
   char *simbol;
   int nrat;
} element;

typedef struct {
    char *nume;
    char *snume;
    int nel;
    element chimism[10];
} bolovan;
```

```
Nume: Smarald
Nume in changestruct: Diamant
Nume: Smarald
```

https://onlinegdb.com/BkGSTTwc4

Trimiterea structurilor către funcții - by value

struct_changestruct_shallow.c

```c
#include <stdio.h>
#include <string.h>

typedef struct IntPtrs {
   int *p1;
   int *p2;
} pointeri;

void changestruct (pointeri pr);

int main() {
   int fvar = 12;
   int *pfvar = &fvar;
   int svar = 123;
   int *psvar = &svar;

   pointeri POI1, POI2;
   POI1.p1 = pfvar;
   POI1.p2 = psvar;
   printf ("Ad POI1.p1 = %p\n", POI1.p1);
   printf ("Ad POI1.p2 = %p\n", POI1.p2);
   POI2 = POI1;
   changestruct(POI2);
}

void changestruct (pointeri pr) {
   printf ("changestruct: Ad POI2.p1 = %p\n", pr.p1);
   printf ("changestruct: Ad POI2.p2 = %p\n", pr.p2);

}
```

Copia structurii trimise by value este shallow. Dacă structura conține pointeri, adresele pe care le stochează aceștia vor rămâne aceleași.

```
Ad POI1.p1 = 0x7ffd0661e7a8
Ad POI1.p2 = 0x7ffd0661e7ac
changestruct: Ad POI2.p1 = 0x7ffd0661e7a8
changestruct: Ad POI2.p2 = 0x7ffd0661e7ac
```

https://onlinegdb.com/BJlq6pPq4

Trimiterea referințelor către funcții - by reference

Acest mod trimite de fapt doar adresa (referința) către începutul spațiului de memorie unde este stocată structura. Funcția îl primește într-un pointer și modificările sunt făcute direct pe structura originală. Observați utilizarea operatorului (->) în funcție, în loc de (.).

struct_changestruct_pointer.c

```c
#include <stdio.h>
#include <string.h>

void changestruct(bolovan * piatra);

int main()
{
   element eal = { "Aluminiu", "Al", 13};
   element eo = { "Oxigen", "O", 8};
   element esi = { "Siliciu", "Si", 14};
   element ebe = { "Beriliu", "Be", 4};
   element ec = { "Carbon", "C", 6};
   bolovan smarald = { "Smarald", "Beril", 4, {ebe, eal,esi,eo}};
   printf ("Nume: %s\n", smarald.nume);
   changestruct(&smarald);
   printf ("Nume: %s\n", smarald.nume);
}

void changestruct (bolovan * piatra) {
   piatra->nume = "Diamant";
   printf ("Nume in changestruct: %s\n", piatra->nume);
}
```

```c
typedef struct {
   char *nume;
   char *simbol;
   int nrat;
} element;

typedef struct {
    char *nume;
    char *snume;
    int nel;
    element chimism[10];
} bolovan;
```

```
Nume: Smarald
Nume in changestruct: Diamant
Nume: Diamant
```

https://onlinegdb.com/BJJECTP5V

Trimiterea șirurilor de structuri către funcții

https://onlinegdb.com/HkjOCTDcE

```c
#include <stdio.h>
#include <string.h>

void changestruct(bolovan * piatra[], int count);

int main()
{
   element eal = { "Aluminiu", "Al", 13};
   element eo = { "Oxigen", "O", 8};
   element esi = { "Siliciu", "Si", 14};
   element ebe = { "Beriliu", "Be", 4};
   element ec = { "Carbon", "C", 6};
  bolovan stones[] = {{ "Rubin", "Corindon", 2, {eal, eo}},
            { "Smarald", "Beril", 4, {ebe, eal,esi,eo}}};

   printstruct(stones, 2);
}

void printstruct (bolovan piatra[], int count) {
  for (int j=0; j<count; j++) {
     printf ("Nume: %s\n", piatra[j].nume);
     printf ("Nume stiintific: %s\n", piatra[j].snume);
     for (int i = 0; i < piatra[j].nel; i++) {
       printf ("--->%s Z: %d\n", piatra[j].chimism[i].nume, piatra[j].chimism[i].nrat);
     }
  }
}
```

```c
typedef struct {
   char *nume;
   char *simbol;
   int nrat;
} element;

typedef struct {
    char *nume;
    char *snume;
    int nel;
    element chimism[10];
} bolovan;
```

struct_array_function.c

```
Nume: Rubin
Nume stiintific: Corindon
--->Aluminiu Z: 13
--->Oxigen Z: 8
Nume: Smarald
Nume stiintific: Beril
--->Beriliu Z: 4
--->Aluminiu Z: 13
--->Siliciu Z: 14
--->Oxigen Z: 8
```

Trimiterea șirurilor de structuri către funcții se face la fel ca trimiterea oricărui alt șir, se declară funcția corespunzător și se adaugă dimensiunea șirului la parametrii.

Returnarea structurilor din funcții

Returnarea structurilor din funcții se face la fel ca returnarea oricărui al tip de variabilă simplă. Funcția trebuie declarată cu tipul structurii respective.

```c
#include <stdio.h>
#include <string.h>

bolovan fillstruct();

int main()
{
   bolovan smarald = fillstruct();
   printf ("Nume: %s\n", smarald.nume);
   printf ("Nume stiintific: %s\n", smarald.snume);
   for (int i = 0; i < smarald.nel; i++) {
       printf ("--->%s Z: %d\n", smarald.chimism[i].nume, smarald.chimism[i].nrat);
   }
}

bolovan fillstruct () {
   element eal = {"Aluminiu", "Al", 13};
   element eo = {"Oxigen", "O", 8};
   element esi = {"Siliciu", "Si", 14};
   element ebe = {"Beriliu", "Be", 4};
   bolovan piatra = {"Smarald", "Beril", 4, {ebe, eal,esi,eo}};
   return piatra;
}
```

```c
typedef struct {
   char *nume;
   char *simbol;
   int nrat;
} element;

typedef struct {
    char *nume;
    char *snume;
    int nel;
    element chimism[10];
} bolovan;
```

```
Nume: Smarald
Nume stiintific: Beril
--->Beriliu Z: 4
--->Aluminiu Z: 13
--->Siliciu Z: 14
--->Oxigen Z: 8
```

struct_return.c

https://onlinegdb.com/r1hs06wc4

Union, Enum

Union

O uniune este un tip compus de dată în C care poate fi definită cu mai mulți posibili membrii (similar unei structuri) dar care nu poate conține decât o singură valoare (doar unul dintre membrii este definit), pentru că datele sunt stocate în aceeași locație de memorie.

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


union Data {
  int i;
  float f;
  char str[20];
};


int main()  {

   union Data rezultat;
   printf ("Memoria ocupata: %ld\n", sizeof(rezultat));
   rezultat.i = 21;
   printf ("Rezultat i: %d\n", rezultat.i);
   rezultat.f = 21.232;
   printf ("Rezultat f: %f\n", rezultat.f);
   strcpy(rezultat.str, "Nu stiu ce sa scriu");
   printf ("Rezultat str: %s\n", rezultat.str);
   printf ("Rezultat i: %d\n", rezultat.i);
   return 0;
}
```

```
Memoria ocupata: 20
Rezultat i: 21
Rezultat f: 21.232000
Rezultat str: Nu stiu ce sa scriu
Rezultat i: 1931507022
```

Memoria ocupată este egală cu memoria ocupată de cel mai voluminos membru

După atribuirea valorii “Nu stiu ce sa scriu” membrului de tip șir de caractere, nu se mai poate accesa nici unul din membrii anteriori (valorile sunt greșite)

union.c

Union este util, în special ca membru al unor structuri mai complexe, în cazuri în care datele reale nu se supun tipurilor C - de exemplu așteptăm numere dar nu știm dacă vor fi întregi lungi sau scurte, float-uri, etc.

https://onlinegdb.com/ryncIsRFN

Union peste struct

```
#include <stdio.h>


typedef union
{
 struct
 {
   unsigned char bit8 : 1;
   unsigned char bit7 : 1;
   unsigned char bit6 : 1;
   unsigned char bit5 : 1;
   unsigned char bit4 : 1;
   unsigned char bit3 : 1;
   unsigned char bit2 : 1;
   unsigned char bit1 : 1;
 }u;
 char status;
}DeviceStatus;


int main()
{
   DeviceStatus ds;
   ds.status = 121;
   printf("bits: %d %d %d %d %d %d %d %d
",ds.u.bit1,ds.u.bit2,ds.u.bit3,ds.u.bit4,ds.u.bit5,ds.u.bit6,ds.u.bit7,ds.u.bit8 );
}
```

char status și structura numită u împart aceeași memorie. Astfel, dacă inițializăm ds.status cu o valoare corespunzătoare (121), aceiași biți utilizați pentru stocarea variabilei de tip caracter “status” pot fi accesați de structura u care are 8 membri a câte un bit.

Această tehnică se folosește foarte mult în programarea registrelor hardware.

Tehnica se bazează pe faptul că în C putem specifica dimensiunea în biți a membrilor structurilor și a uniunilor. Aceasta se face prin folosirea semnului :

unsigned char bit8 : 1 - definește folosirea unui singur bit

Acest mod de definire se numește “bit field”.

Această tehnică este extrem de utilizată în “despachetarea” rapidă a multor date în biți sau în fragmente de alte dimensiuni (registre hardware, culori, adrese ip, etc.). în sursa nucleului Linux cuvântul union este prezent de 3032 de ori

```
bits: 0 1 1 1 1 0 0 1
```

https://onlinegdb.com/SkHSL3AYV

union_struct.c

Enum

enum este un tip de date care crează un alias între cuvinte și numere întregi. Este folosit în special pentru a face codul sursă mai ușor de citit.

Enum înlocuiește numerele cu cuvintele.

```c
#include <stdio.h>

enum year{Jan, Feb, Mar, Apr, May, Jun,
Jul, Aug, Sep, Oct, Nov, Dec};

int main()
{
  int i;
  enum year ch;
  ch = May;
  for (i=Jan; i<=Dec; i++) {
     printf("%d ", i);
  }
  printf ("\nch este: %d\n",ch);
  return 0;
}
```

enum_1.c

```
0 1 2 3 4 5 6 7 8 9 10 11
ch este: 4
```

```c
#include <stdio.h>

enum year{Jan=10, Feb, Mar, Apr, May,
Jun, Jul, Aug, Sep, Oct, Nov, Dec};

int main()
{
  int i;
  enum year ch;
  ch = May;
  for (i=Jan; i<=Dec; i++) {
     printf("%d ", i);
  }
  printf ("\nch este: %d\n",ch);
  return 0;
}
```

enum_init.c

Etichetele din enum pot fi inițializate parțial sau total.

```
10 11 12 13 14 15 16 17 18 19 20 21
ch este: 14
```

https://onlinegdb.com/ryYIc30KE

Liste înlănțuite

Structurile permit gruparea mai multor tipuri de date

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

struct Stone
   {
        char nume[50];
        char snume[50];
        char formula[50];
        int duritate;
        float refr;
        char foto[30];
        char clr[30];
   };


int main()

{
   struct Stone rubin, safir, smarald, diamant;

   strcpy(rubin.nume, "Rubin");
   strcpy(rubin.snume, "Corindon");
   strcpy(rubin.formula, "Al2O3");

   printf ("Nume: %s\n", rubin.nume);
   printf ("Nume stiintific: %s\n", rubin.snume);
   printf ("Formula chimica: %s\n", rubin.formula);

}
```

Structurile se definesc de obicei în afara funcției main() pentru a avea un scop global.

Structurile sunt tipuri complexe de date, nu grupări de variabile. După definirea lor, trebuiesc declarate variabile cu noul tip.

Accesarea membrilor unei structuri (citire și scriere) se face cu ajutorul operatorului punct (.)

```
Nume: Rubin
Nume stiintific: Corindon
Formula chimica: Al2O3
```

Liste înlănțuite

Listele înlănțuite sunt implementări particulare ale structurilor, în care unul dintre membrii (numit nod) este un pointer către o structură identică.

```
#include <stdio.h>

int main() {
   typedef struct node {
       int val;
       struct node * next;
   } node_t;


return 0;
}
```

Structura numită node_t are doi membrii:

int val - un număr întreg numit val

struct node * next - un membru numit next care este un pointer către o structură de tip node.

Cu ajutorul acestui tip special de structură se poate implementa un fel de șir (o colecție unidimensională de obiecte identice) care pot fi grupate împreună și accesate în ordinea în care au fost definite.

Fiecare element al listei conține un pointer către următorul element, deci fiecare element știe care este succesorul său. De aceea, listele înlănțuite sunt ordonate.

Elementele unui șir definit ca listă înlănțuită nu trebuie sa fie succesive în memorie.

https://www.learn-c.org/en/Linked_lists

Liste înlănțuite

linked_list_1.c

```c
typedef struct node {
    int val;
    struct node * next;
} node_t;
```

```c
int main()
{
    node_t * head = NULL;
    head = malloc(sizeof(node_t));
    if (head == NULL) {
        return 1;
    }
    head->val = 1;
    head->next = NULL;

/* al doilea nod */
    head->next = malloc(sizeof(node_t));
    head->next->val = 2;
    head->next->next = NULL;

/* al treilea nod */
    head->next->next = malloc(sizeof(node_t));
    head->next->next->val = 3;
    head->next->next->next = NULL;
}
```

Definirea structurii de tip listă înlănțuită

Pointer către o structură de tipul node_t

Alocă spațiu pentru un element al listei

malloc returnează NULL dacă nu reușește să aloce memoria. Funcția main() poate returna o valoare nenulă dacă programul nu a rulat corect.

Setăm valoarea membrului head ca fiind 1.

Prin setarea pointerului next ca NULL ne asigurăm că head este ultimul membru al listei

introducerea celui de-al doilea termen al listei

introducerea celui de-al treilea termen al listei

https://onlinegdb.com/LO4tsP3Nw

Liste înlănțuite

linked_list_2.c

```
int main()
{
  node_t * head = NULL;
  head = malloc(sizeof(node_t));
  if (head == NULL) {
      return 1;
  }
  head->val = 1;
  head->next = NULL;
/* al doilea nod */
  head->next = malloc(sizeof(node_t));
  head->next->val = 2;
  head->next->next = NULL;
/* al treilea nod */
  head->next->next = malloc(sizeof(node_t));
  head->next->next->val = 3;
  head->next->next->next = NULL;

  printf("dim struct: %ld\n", sizeof(node_t));
  printf("head addr: %p\n", &head);
  printf("head: %p\n", head);
  printf("head val: %d\n", head->val);
  printf("head next: %p\n", head->next);
  printf("head->next val: %d\n", head->next->val);
  printf("head->next next: %p\n", head->next->next);
  printf("head->next->next val: %d\n", head->next->next->val);
  printf("head->next->next next: %p\n", head->next->next->next);
}
```

```
dim struct: 16
head addr: 0x7ffe66e4fe00
head: 0x55dc07e3d260
head val: 1
head next: 0x55dc07e3d280
head->next val: 2
head->next next: 0x55dc07e3d2a0
head->next->next val: 3
head->next->next next: (nil)
```

https://onlinegdb.com/B1LY7wAt4

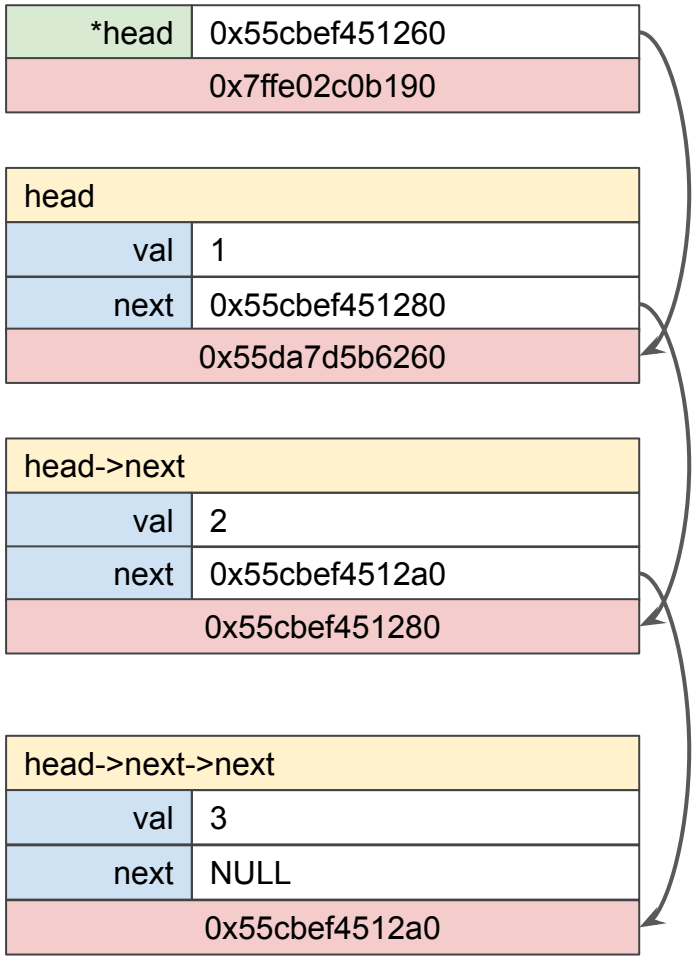
Liste înlănțuite simple

```
dim struct: 16
head addr: 0x7ffe66e4fe00
head: 0x55dc07e3d260
head val: 1
head next: 0x55dc07e3d280
head->next val: 2
head->next next: 0x55dc07e3d2a0
head->next->next val: 3
head->next->next next: (nil)
```

Listele înlănțuite simple nu pot fi parcurse decât de la stânga la dreapta (de la head spre final).

variabila head va fi stocată în stack, dar ea nu este decât un pointer către primul element al listei. Celelalte elemente ale listei vor fi definite în heap, una după alta pentru că așa sunt inițializate în program.

Continuitatea în memorie a membrilor unei liste înlănțuite nu este niciodată garantată.



Traversarea unei liste înlănțuite simple

Traversarea unei liste înlănțuite se face cu ajutorul unui pointer temporar care va conține rând pe rând fiecare nod al listei. Acest pointer se mai numește și “walker”.

```
void print_list(node_t * head) {

   node_t * current = head;

    while (current != NULL) {
        printf("%d\n", current->val);
        current = current->next;
    }
}
```

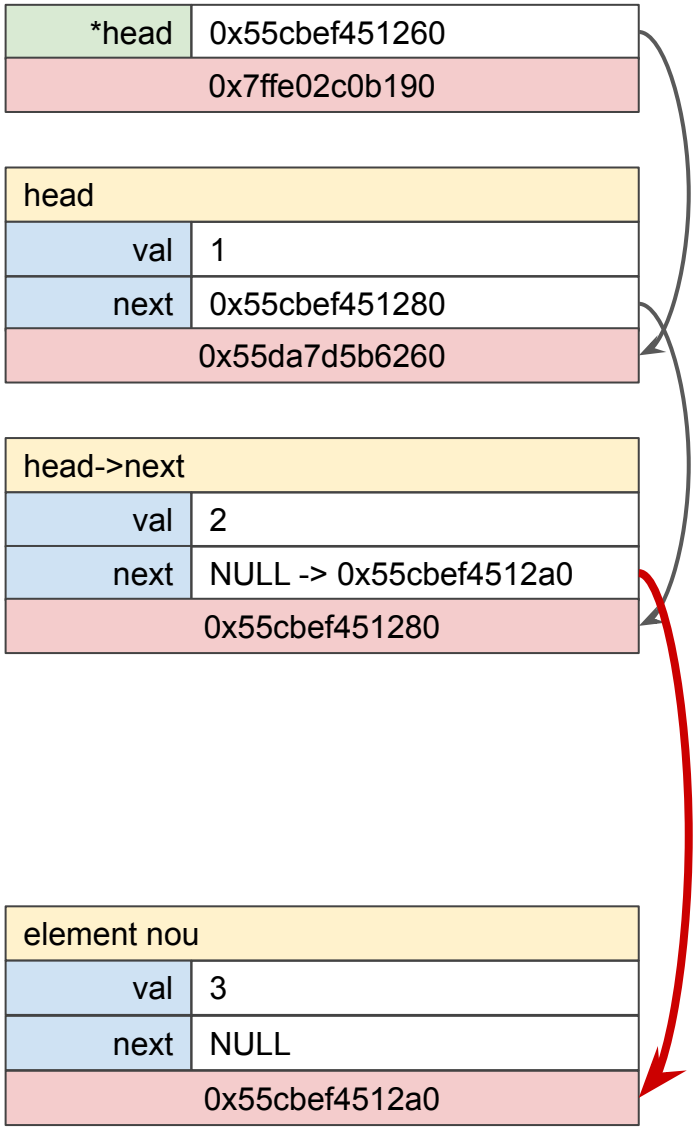
Pointer către o structură de tipul node_t care va trimite succesiv către fiecare nod al listei (walker)

Testează pentru a vedea dacă pointerul curent nu este NULL (finalul listei)

Afișează valoarea elementului curent

Atribuie walkerului următorul element al listei

Liste înlănțuite simple - push pe ultima pozitie



Push pe ultima poziție presupune:

1. Parcurgerea listei de la primul membru până la ultimul
2. Alocăm memorie pentru ultimul membru
3. Facem legătura între pointerul next al ultimului membru (înlocuind NULL cu adresa noului element).

Liste înlănțuite simple - push pe ultima poziție

Funcția push adaugă un membru la sfârșitul listei înlănțuite.

```
push (head, 12);
```

```
int push(node_t * head, int val) {
    node_t * current = head;
    while (current->next != NULL) {
        current = current->next;
    }

    current->next = malloc(sizeof(node_t));
    if (head == NULL) {
      return -1;
    }
    current->next->val = val;
    current->next->next = NULL;
    return 1;
}
```

ll_operations.c

https://onlinegdb.com/njsX56frk

Declarăm o variabilă intermediară (un pointer de tipul node_t) care va ține elementul curent al listei

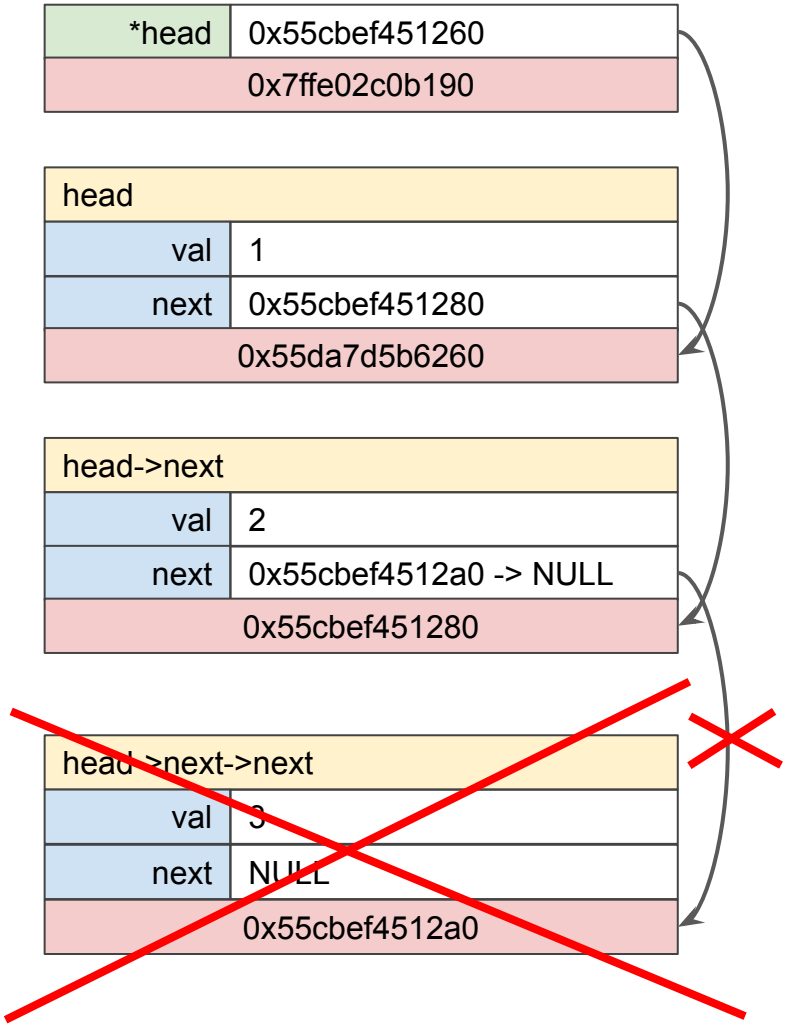
Parcurgem lista element cu element până când ajungem la ultimul membru (cel care are pointerul NULL). Atribuim fiecare element variabilei current. La finalizarea buclei, variabila current va deține un pointer către ultimul element al listei

alocăm memorie pentru noul membru al listei

Verificăm dacă malloc-ul a reușit să aloce memoria

Atribuim valoarea corespunzătoare și pointerul NULL noului membru al listei

Liste înlănțuite simple - pop



Parcurgem lista inlănțuită până la penultimul element (ultimul care are pointerul catre next nenul

Eliberăm memoria ocupată de ultimul element al listei

Atribuim pointerul NULL membrului next al elementului curent al listei, care va deveni astfel ultimul

Liste înlănțuite simple - pop

Funcția pop returnează și elimină ultimul membru al listei.

```
int rval = pop(head);
```

```
int pop(node t * head) {
    int retval = 0;
    if (head->next == NULL) {
        retval = head->val;
        free(head);
        return retval;
    }
    node t * current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }
    retval = current->next->val;
    free(current->next);
    current->next = NULL;
    return retval;
}
```

ll_operations.c

https://onlinegdb.com/njsX56frk

free nu este suficientă pentru că nu golește memoria. O face disponibilă pentru un eventual malloc următor dar pointerul continuă să trimită către datele existente. De aceea este nevoie de atribuirea pointerului NULL

Declarăm o variabilă cu ajutorul căreia vom putea returna valoarea scoasă

Dacă lista are un singur element (adică head->next este NULL) ștergem elementul și eliberăm memoria.

Declarăm o variabila pentru nodul curent (walker)

Parcurgem lista înlănțuită până la penultimul element (ultimul care are pointerul către next nenul

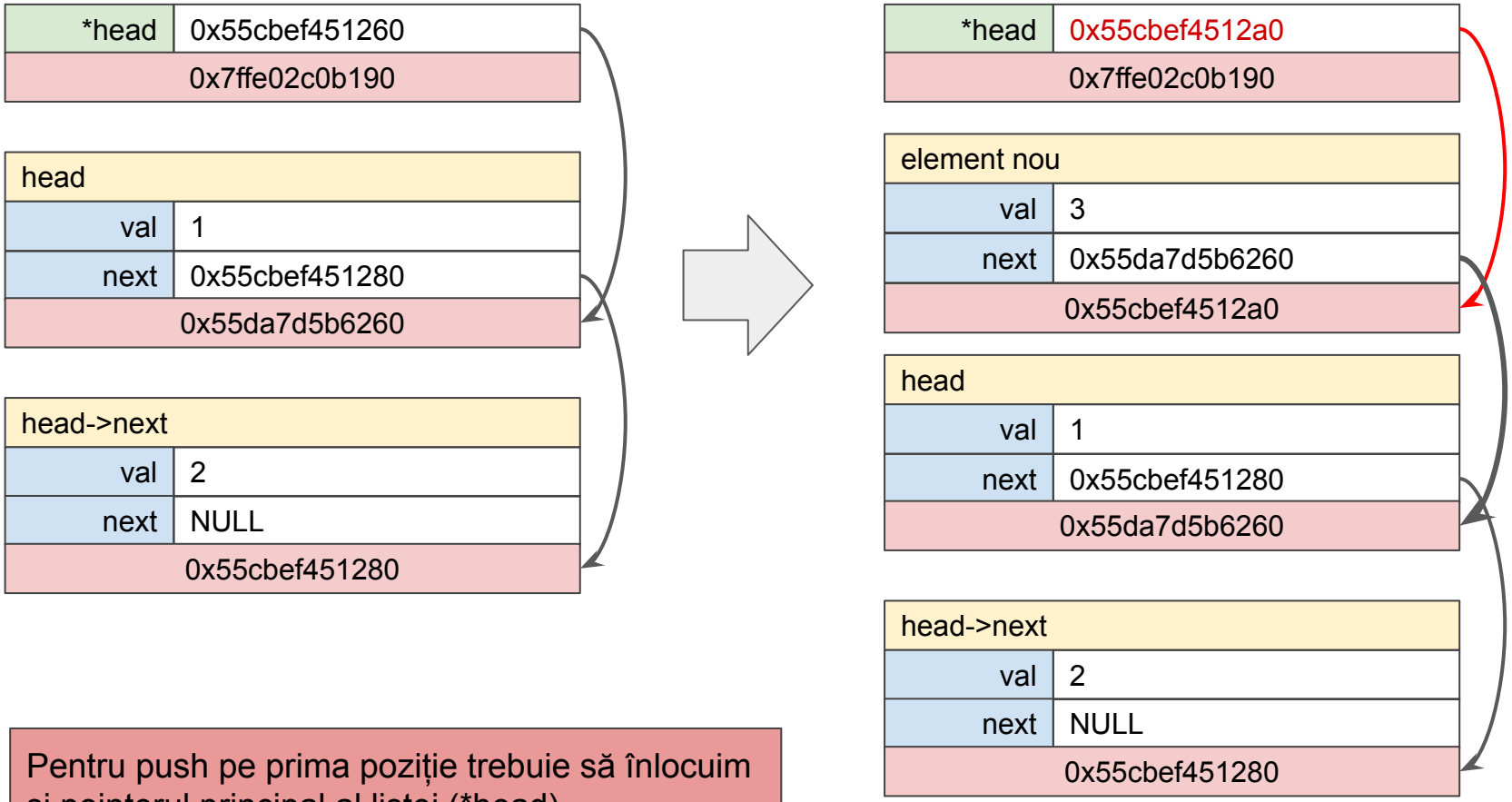
Atribuim variabilei retval valoarea următorului membru al listei (cel care va fi șters)

Eliberăm memoria ocupată de ultimul element al listei

Atribuim pointerul NULL membrului next al elementului curent al listei, care va deveni astfel ultimul

returnăm valoarea ștearsă (retval)

Liste înlănțuite simple - push pe prima pozitie



Pentru push pe prima poziție trebuie să înlocuim și pointerul principal al listei (*head).

Liste înlănțuite simple - push pe prima pozitie

Functia f_push adaugă un membru la începutul listei înlănțuite.

```
int rval = f_push (&head, 21);
```

```
int f_push(node_t ** head, int val) {
    node_t * new_node;
    new_node = malloc(sizeof(node_t));
    if (head == NULL) {
        return -1;
    }
    new_node->val = val;
    new_node->next = *head;
    *head = new_node;
    return 1;
}
```

ll_operations.c

Declarăm un pointer de tip node_t către noul nod al listei

Alocăm memorie pentru noul membru al listei

Verificăm dacă malloc a reușit să aloce memoria

setăm valoarea primită pentru noul membru al listei

Atribuim pointerul către head (care este primul element al listei înainte de execuția funcției curente) membrului next al listei

Atribuim noul nod pointerului head, noul nod devenind astfel capătul listei

Această operațiune ne obligă ca la intrarea în funcție să trimitem un pointer către pointer. Dacă am fi modificat doar variabila către care pointerul trimite, ar fi fost suficient un pointer simplu dar noi modificam chiar pointerul, făcându-l să trimită către altă adresa, deci avem nevoie de un pointer către el.

https://onlinegdb.com/njsX56frk

Liste înlănțuite simple - f_pop prima pozitie

Functia f_pop șterge primul element al listei înlănțuite

```
int rval = f_pop(&head);
```

```
int f_pop(node_t ** head) {
    int retval = -1;
    node_t * next_node = NULL;

    if (*head == NULL) {
        return -1;
    }

    next_node = (*head)->next;
    retval = (*head)->val;
    free(*head);
    *head = next_node;
    return retval;
}
```

ll_operations.c

Initializăm o variabilă în care vom pune valoarea pe care o returnăm

Definim o nouă variabilă de tip node_t pentru viitorul head

Dacă *head == NULL înseamnă că lista e goală

atribuim variabilei next_node celui de-al doilea nod al listei

Atribuim valoarea *head->val (*head va fi șters) variabilei retval

Eliberăm memoria ocupată de primul element al listei

Atribuim variabila next_node pointerului *head

Returnăm retval

Aceasta operațiune ne obligă ca la intrarea în funcție să trimitem un pointer către pointer. Dacă am fi modificat doar variabila către care pointerul trimite, ar fi fost suficient un pointer simplu dar noi modificam chiar pointerul, făcându-l să trimită către alta adresă, deci avem nevoie de un pointer către el.

https://onlinegdb.com/njsX56frk

Liste înlănțuite - sterge un index

```
int rval = f_pop(&head);
```

```
int remove_by_index(node_t * head, int n)
{
    int i = 0;
    int retval = -1;
    node_t * current = head;
    node_t * temp_node = NULL;
    if (n == 0) {
        return f_pop(head);
    }
    for (i = 0; i < n-1; i++) {
        if (current->next == NULL) {
            return -1;
        }
        current = current->next;
    }
    temp_node = current->next;
    retval = temp_node->val;
    current->next = temp_node->next;
    free(temp_node);
    return retval;
}
```

ll_operations.c

Initializam o variabila pentru ciclare

Inițializăm o variabilă în care vom întoarce valoarea eliminată

Inițializăm o variabilă de tip node_t căruia îi dăm valoarea primului element al listei

Inițializăm o variabilă de tip node_t cu NULL

dacă se solicită eliminarea primului element (n=0) atunci vom folosi funcția f_pop

Parcurgem primele n-1 elemente ale listei pentru a ne poziționa corect. La final, variabila current conține pointerul către elementul care trebuie șters

Atribuim variabilei temp_node pointerul către nodul care va fi șters

Punem deoparte valoarea pe care o vom șterge

Mutam pointerul current->next un element mai departe

Eliberăm memoria ocupată de temp_node

https://onlinegdb.com/njsX56frk

Semnificația octeților

Biții au, câteodată, semnificație

MSB - most significant bit

least significant bit - LSB

0	1	1	1	1	0	0	1
*	*	*	*	*	*	*	*
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$0 + 64 + 32 + 16 + 8 + 0 + 0 + 1 = 121$

MSB - Bitul cel mai semnificativ este cel care, modificat, are cel mai mare efect asupra informației stocate.

LSB - Bitul cel mai puțin semnificativ este cel care, modificat, are cel mai mic efect asupra informației stocate.

Efectul modificării LSB este atât de mic încât poate fi înlocuit complet fără efecte vizibile în imagini. Această tehnică se numește steganografie.

Aceasta imagine (512 x 365 pixeli) contine circa 36 de pagini de text, ascunse prin inlocuirea celor mai puțini semnificativi biți cu cei ai mesajului (~ 500k octeți)

https://stylesuxx.github.io/steganography/

În cazul numerelor stocate în mai mulți octeții vor avea de asemenea semnificații diferite

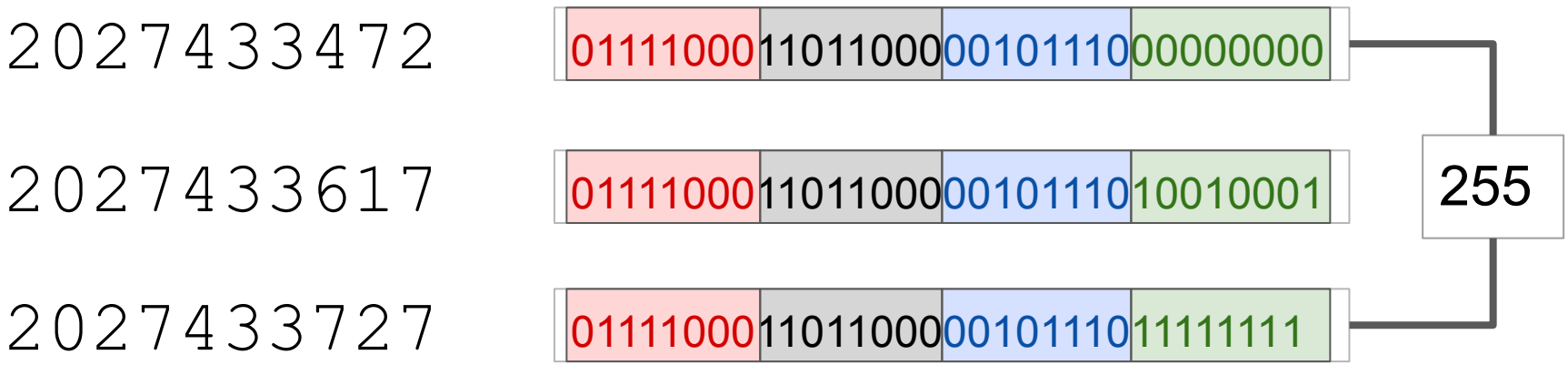
2027433617

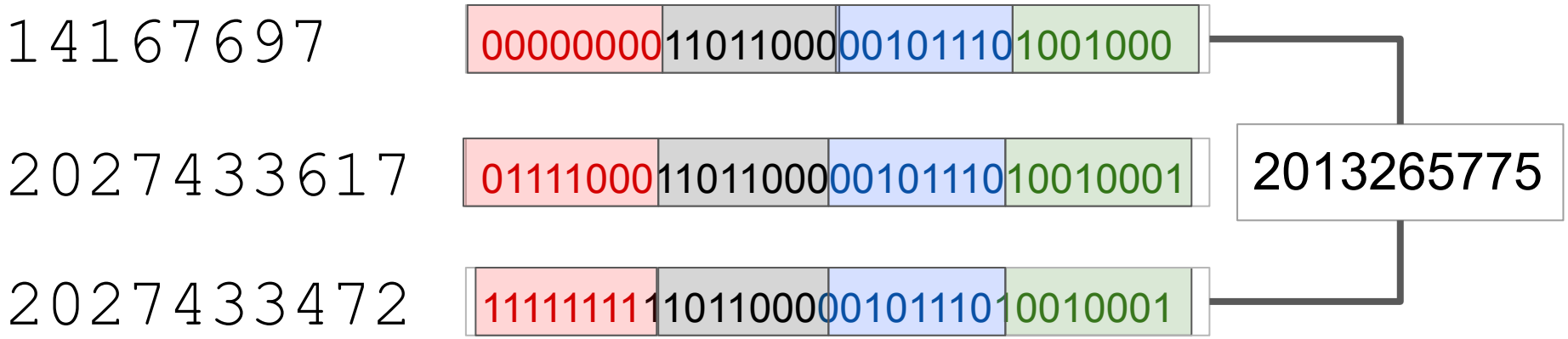
01111000	11011000	00101110	10010001

Cel mai semnificativ octet

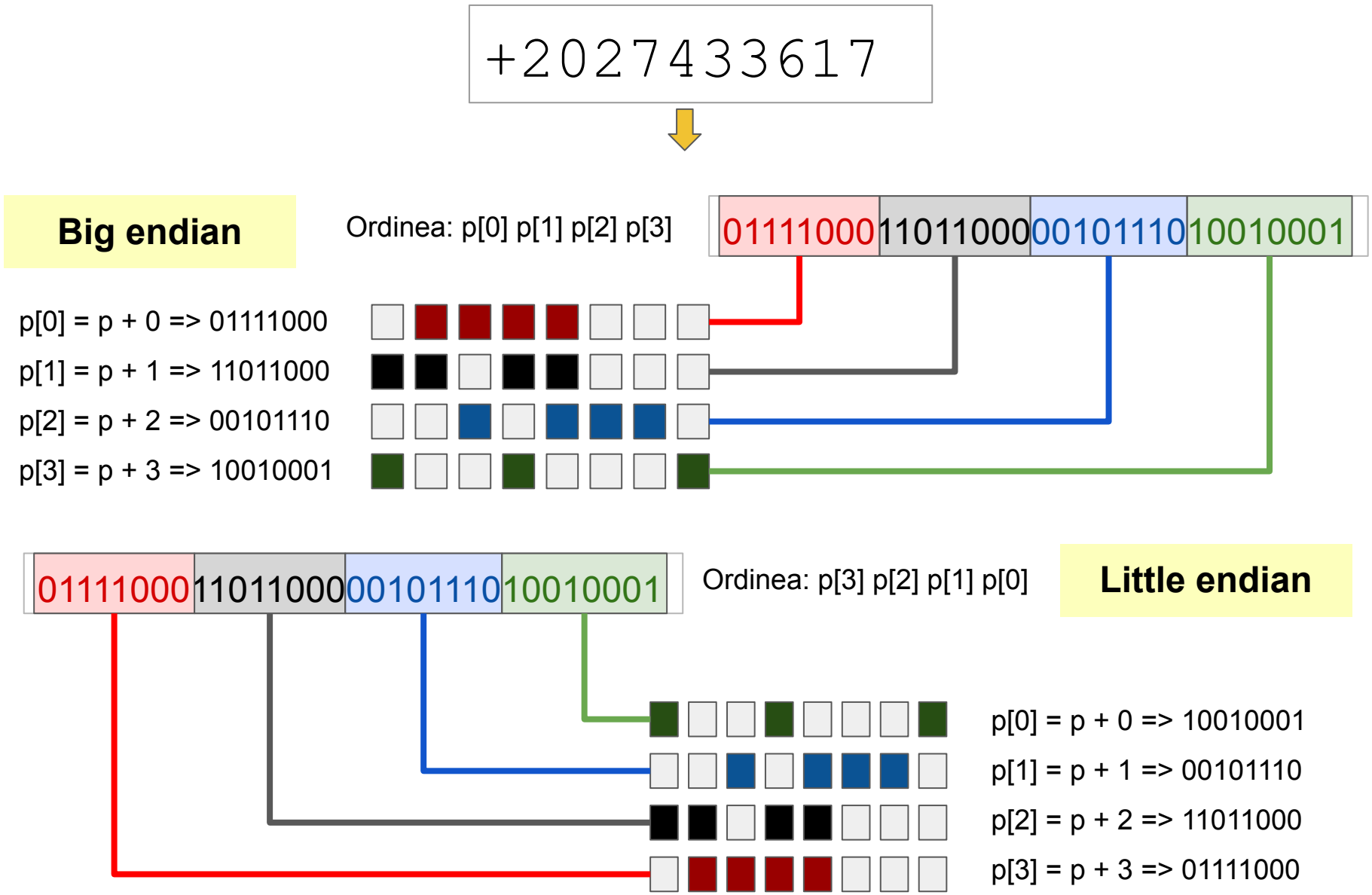
Cel mai puțin semnificativ octet

Cel mai semnificativ octet și cel mai puțin semnificativ octet





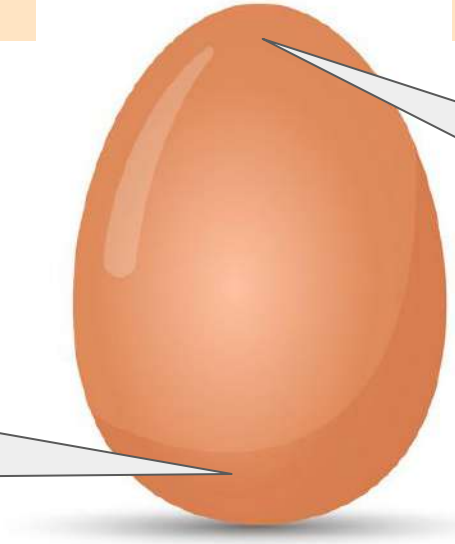
Octeții se pot stoca în memorie în două moduri



Care este modul acceptabil de a sparge un ou?

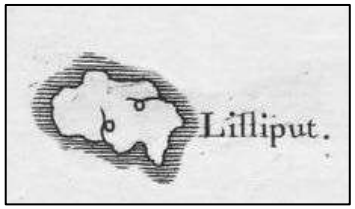
Big Endians

Little Endians

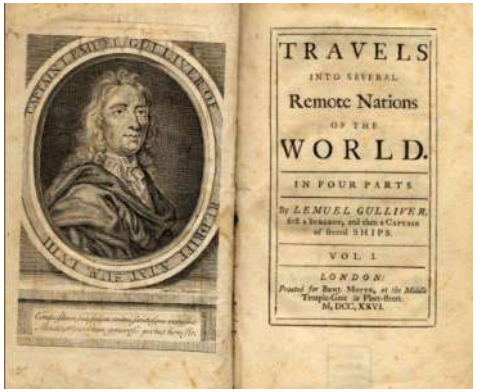


little end

big end



Jonathan Swift



Călătoriile lui Guliver, 1726

Dany Cohen

http://www.networksorcery.com/enp/ien/ien137.txt

Octeții se pot stoca în memorie în două moduri - De ce?

Aproape toate arhitecturile au rădăcini în computere care puteau citi doar pachete de 8 biți. În acele cazuri, fiecare abordare avea avantaje.

Little endian:

- Crește viteza operațiilor de adăugare
- Castingul de la variabile stocate într-un număr mai mare de biți către variabile stocate într-un număr mai mic de biți este mai rapid.

Big endian:

- Permite estimări rapide ale ordinului de mărime al numerelor prin citirea octetului mai semnificativ
- Crește viteza verificării semnului numerelor - acesta se află tot timpul în bitul cel mai semnificativ.

În cazul arhitecturilor moderne (32, 64 biți) acestea nu mai contează pentru că procesorul citește direct toți octeții majorității tipurilor de date

Protocoalele de comunicare între dispozitive trebuie să își stabilească propria ordine de transmitere a octeților

Aproape toate protocoalele de rețea sunt big endian (motiv pentru care acesta se mai numește și network order), cu excepția SMB

Formatele binare de stocare a datelor trebuie de asemenea să stabilească ordinea de stocare a octeților

Little endian

Big endian

Big endian

Big endian

Ambele

Little endian

Little endian

Little endian

PowerPC

Little endian
Big endian

Little endian (90%)
Big endian

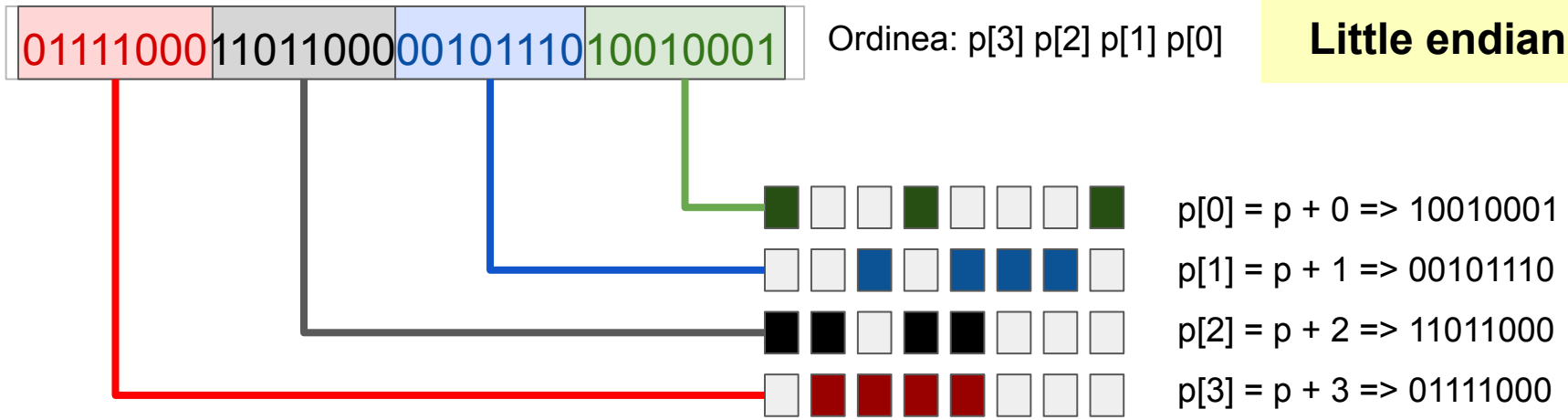
Little endian
Big endian

Big endian

Texas Instruments c2000

Little endian

Citirea unei variabile din memorie



bytes_to_binary.c

```c
int main() {
 int a = 2027433617;
 printf("a este: %i\n", a);
 printf("adresa a: %p, lungime %lu\n", & a, sizeof(a));
 char * p = & a;
 //stack-ul e ordonat invers :):)
 char fb = p[3];
 char sb = p[2];
 char tb = p[1];
 char frb = p[0];
}
```

https://onlinegdb.com/ry5pXgEu4

Little și Big endian se referă la ordinea stocării octeților în memorie. Stocarea biților în interiorul unui octet se numește "bit endianness" și se face de obicei în aceeași ordine ca cea a octeților pe fiecare platformă.

Bit endianness este mai puțin importantă în relația cu memoria pentru că biții nu au propriile adrese - nu pot fi citiți individual. Bit endianness devine important la comunicare.

Încărcarea și execuția programelor în sistemele de operare

Programele pe care le scriem și le compilăm sau le lansăm în execuție cu ajutorul interpretoarelor, sunt în final executate de un sistem de operare.

Chiar și în cazul execuției în cloud, chiar și în modelul "serverless computing", undeva există un sistem de operare care execută un program.

Unul dintre cele mai importante momente în evoluția sistemelor de operare a fost apariția conceptului de "time sharing", evoluat ulterior în "multitasking". Prima lucrare care menționează "time shating" a fost scrisă de Christopher Strachey în anul 1963 iar John McCarthy este probabil cel care este responsabil de prima implementare, încă înainte de apariția lucrării lui Strachey. Printre altele, John McCarthy a fost unul dintre fondatorii domeniului inteligenței artificiale, a inventat limbajul LISP, a inventat conceptul de "garbage collection".

Time-sharing permitea unui sistem de operare să suspende execuția unui program (fără să-l oprească), să se mute la alt program, urmând ca apoi să revină la primul și să îl poată executa din punctul de unde a rămas. La o viteză acceptabilă de execuție, această mutare foarte rapidă dă senzația execuției paralele a programelor multiple în același timp.

Sistemele de operare fac aceasta de multă vreme. Recent, motoarele de execuție a anumitor medii de programare (ex: Mașina virtuală Java, executoarele PhP, Python, anumite implementări javascript) au început să permită modele de execuție similare permițând extinderea paradigmei procedurale.

Execuția programelor în sistemul de operare

- Un program este un fișier binar care are un format pe care atât nucleul sistemului de operare cât și procesorul îl pot înțelege.
- Nucleul sistemului de operare este responsabil de “încărcarea” programului în memorie - încărcare care implică maparea anumitor zone ale programului în memoria virtuală a procesului respectiv.
- Printre informațiile încărcate de nucleul sistemului de operare în memorie se află și seturi de instrucțiuni destinate procesorului. Acestea trebuie să fie compatibile cu setul de instrucțiuni care pot fi executate de către procesor (ISA).
- Procesorul va începe să execute instrucțiunile încărcate de nucleu în memorie, una câte una, până când acestea se vor termina, apoi procesul se va închide și memoria va fi eliberată.
- **Pentru ca un program să poată fi transformat într-un proces trebuie ca el să fie în același timp compatibil cu nucleul sistemului de operare și cu procesorul pe care acesta rulează.**

Transformarea unui program într-un proces

Transformarea unui program într-un proces se face de către un component special al sistemului de operare, numit “loader”.

- Verifică integritatea și compatibilitatea formatului executabil (arhitectura corecta, programul e complet, etc.)
- Solicită altor componente ale sistemului de operare verificarea drepturilor de execuție
- Copiază imaginea programului în memoria principală a sistemului
- Copiază în zone speciale de memorie funcțiile necesare din biblioteci dinamice (dacă este cazul)
- Copiază datele de intrare ale programului (ex: argumente date la linia de comandă) în stack
- Inițializează regiștrii procesorului
- Pornește **execuția funcției principale** a programului

Funcția principală (main)

Majoritatea formatelor executabile moderne au un singur punct de intrare caruia ii este cedat controlul după încărcarea programului în memorie

Anumite limbaje de programare solicita precizarea explicită a punctului de intrare (C, C++, Java, etc.) iar altele, în special limbajele interpretate, inferă punctul de intrare prin utilizarea primei declarații executabile din corpul programului care nu se găsește în definiția unei funcții (perl, php, javascript, ruby, etc.)

```c
#include <stdio.h>

int main() {
   int n = 4, i;
   float num[] = {12.32, 16.21, 10, 18.1};
   float sum = 0.0;
   float avg;

   for (i = 0; i < n; ++i) {
       sum += num[i];
   }

   avg = sum / n;
   printf("Media = %.2f", avg);
   return 0;
}
```

main_func.c

https://onlinegdb.com/eqNcFP7dn

main

```php
<?php

$array = array(12.32, 16.21, 10, 18.1);
$sum = 0;
for ($i = 0; $i < count($array); ++$i) {
   $sum += $array[$i];
}

$average = $sum / count($array);



echo $average;
```

main_func1.php

https://onlinegdb.com/0D85YHdDt

Php oferă funcții specifice pentru matrici care permit scrierea programului într-o variantă mult redusă

```php
<?php

$array = array(12.32, 16.21, 10, 18.1);

$average = array_sum($array)/count($array);

echo $average;
```

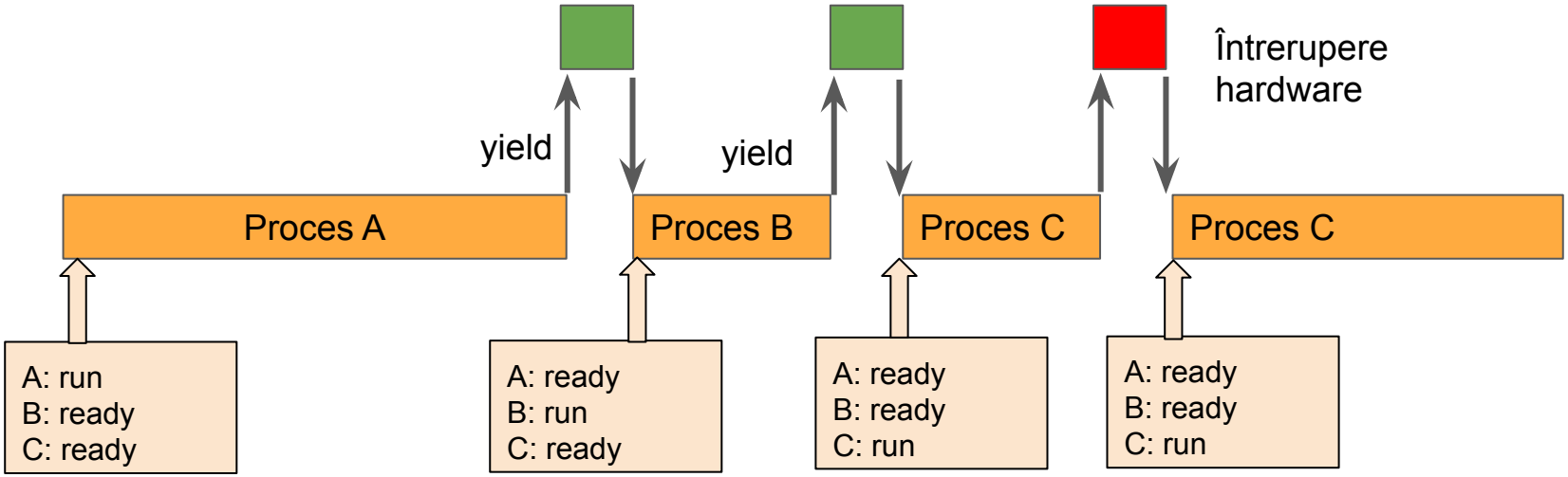
main_func2.php

https://onlinegdb.com/OdKMpx2QaI

Multitasking

Toate sistemele de operare moderne rulează mai multe programe în același timp. Această caracteristică se numește “multitasking”. Multitaskingul implică suspendarea temporară a execuției proceselor cu permiterea rulării unuia dintre ele. Fiecare proces care se află în memorie este rulat din cand in cand. “Semnalul” care declanșează mutarea între procese este dat de întreruperile hardware, provenite de obicei de la ceasul plăcii de bază. Întreruperile pot fi de mai multe tipuri și **au câte o procedură asociată** - această procedură nu face parte din structura vreunui program ci este o succesiune de instrucțiuni predefinite care se execută de câte ori apare acel tip de întrerupere.

Multitasking cooperativ - fiecare proces cedează de bună voie controlul către sistemul de operare

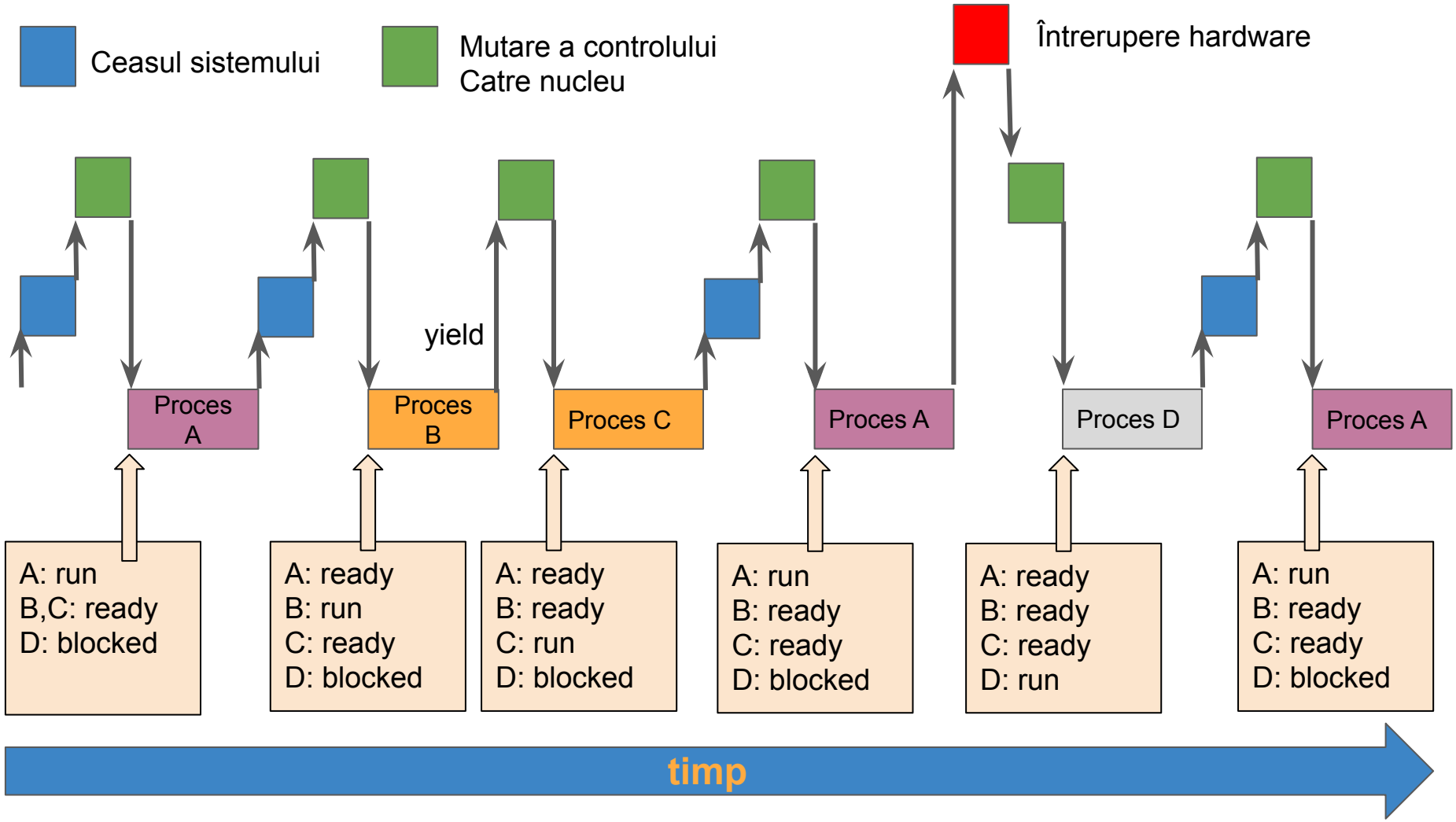


Windows pre 95

Mac OS clasic

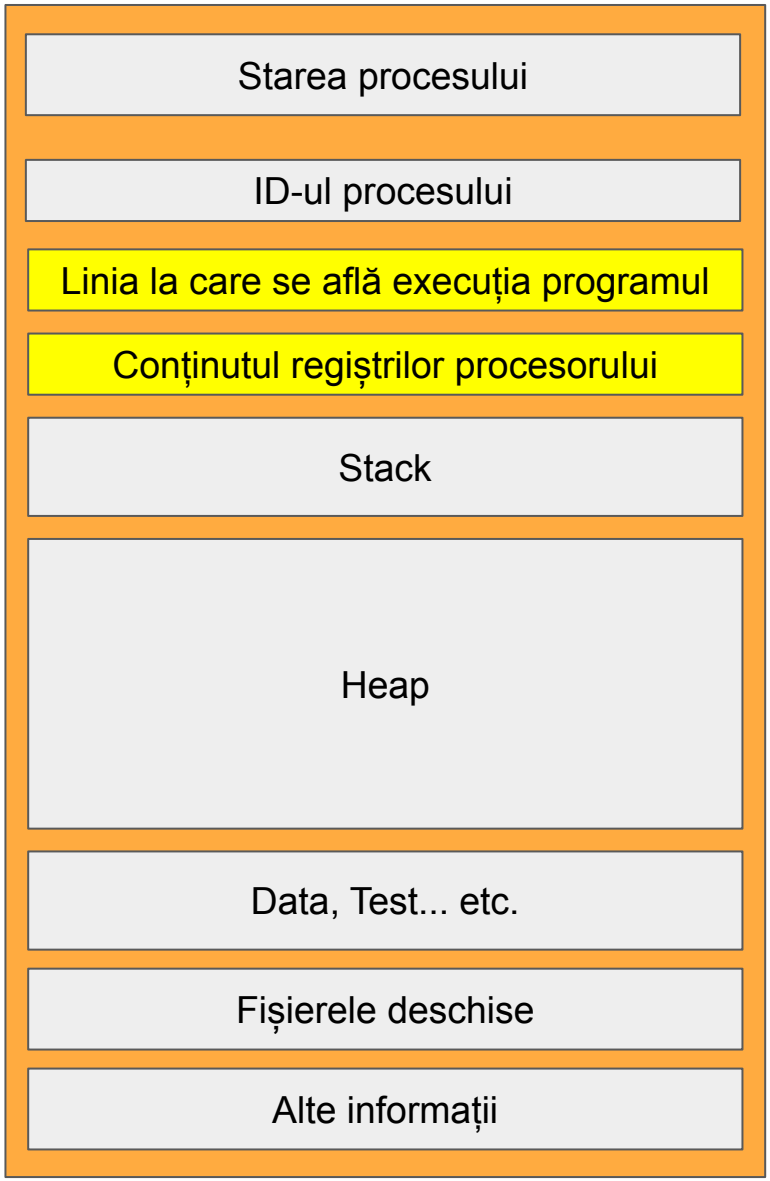
Multitasking preemptiv

Multitaskingul preemptiv forțează suspendarea execuției proceselor și trecerea controlului către nucleu. Procesele pot în continuare să cedeze controlul voluntar dar, dacă nu o fac, vor fi suspendate după un timp predefinit de către algoritmul de planificare

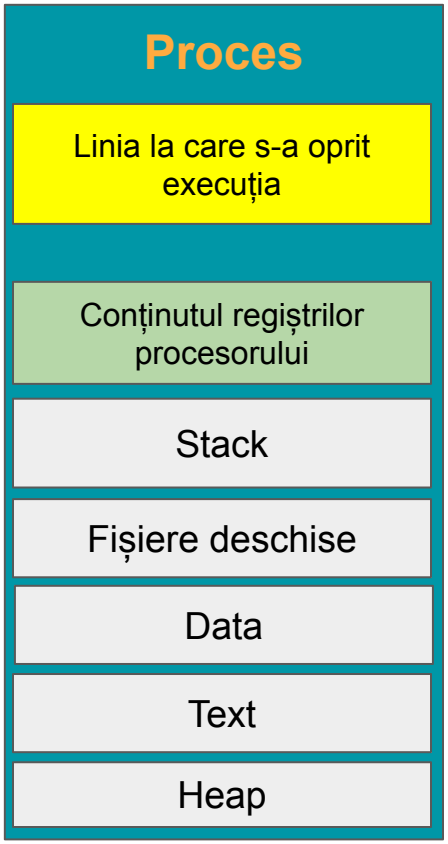


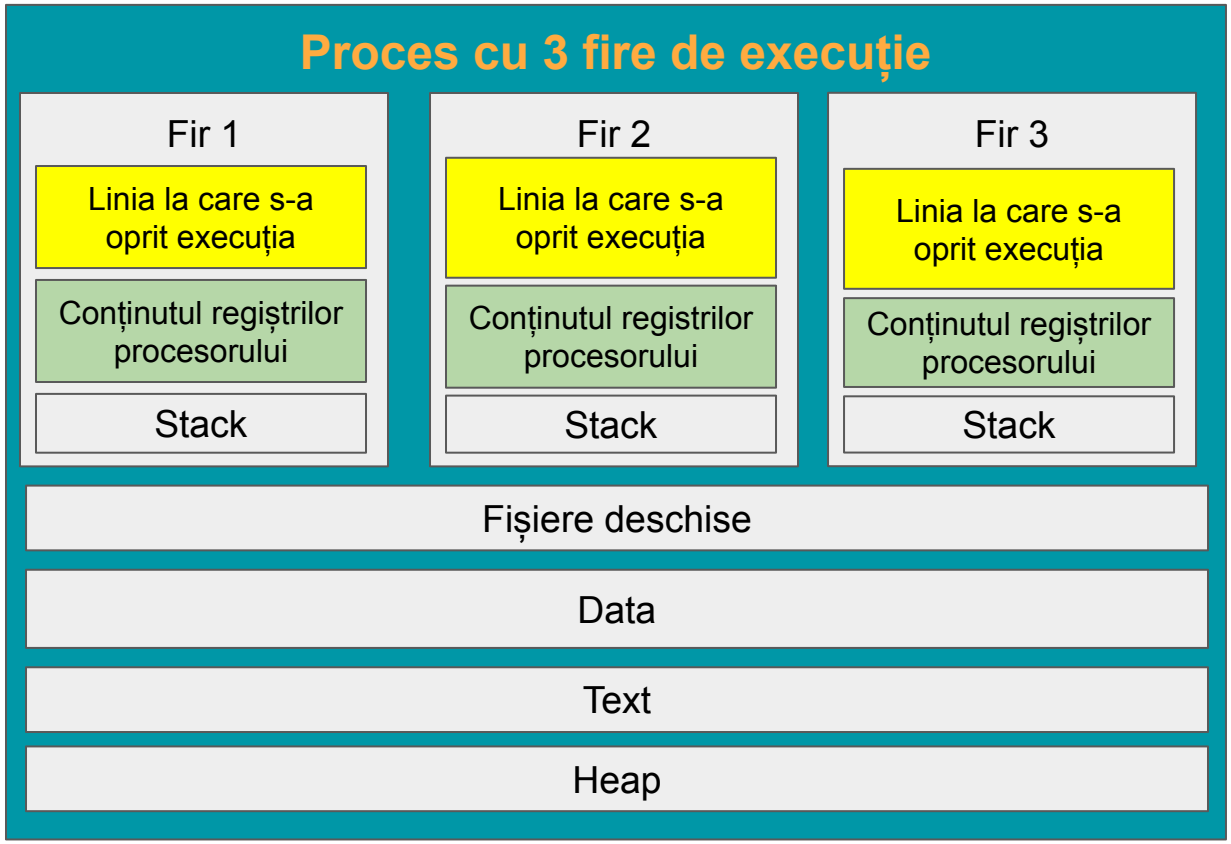
PCB

- La schimbarea stării, datele procesului se salvează în memorie.
- Sistemul conține o serie de containere unde poate salva datele unui proces atunci când acesta este oprit din rulare. Aceste date sunt stocate în structuri de date numite “process control block”.
- PCB-urile NU conțin întreaga memorie ocupată de proces ci liste de referințe.



Fire de execuție (threads) și procese





Procese - instanțe de execuție cu absolut toate segmentele de memorie izolate.
Thread-uri - procese care au mai multe fire de execuție prin separarea stivelor și a conținutului registrelor. Procesul părinte se ocupă de suspendarea/pornirea firelor de execuție. Un proces cu mai multe fire de execuție poate fi considerat un fel de mașină virtuală.

Multitaskingul din interiorul unui proces cu mai multe fire de execuție este de obicei preemptiv. Dacă acesta este cooperativ atunci firele de execuție se mai numesc “**fibre**” (FIBERS).

Un proces (sau un thread) are un singur stack deci poate executa o singură procedură la orice moment!

Un procesor are o singură linie de execuție deci poate executa un singur proces la orice moment!

Un procesor poate optimiza execuția proceselor prin multitasking (cooperativ sau preemptiv). Optimizarea este reală - procesele care așteaptă date sunt puse în așteptare. În final se obține o execuție mai fluidă și mai eficientă a aceleiași liste de procese.

Putem face ceva similar multitaskingului în programare?

Putem defini proceduri care să se execute nu prin apelare directă ci ca răspuns la un anumit eveniment, asociind un set de instrucțiuni acelui eveniment (callbacks, closures).

Putem construi proceduri care să cedeze controlul în timpul execuției (yield) urmând să fie “suspendate” până la următorul apel și să își reia execuția de unde au rămas (generatori).

Putem construi mai multe stack-uri care să constituie fire distincte de execuție în interiorul aceluiași program. În continuare într-un moment dat va exista un singur stack “activ” dar altele pot fi suspendate și reluate atunci când este nevoie (corutine)

Dar modelul de execuție impus de hardware nu permite aceste lucruri

Nu toate programele se compilează direct pentru procesor!!

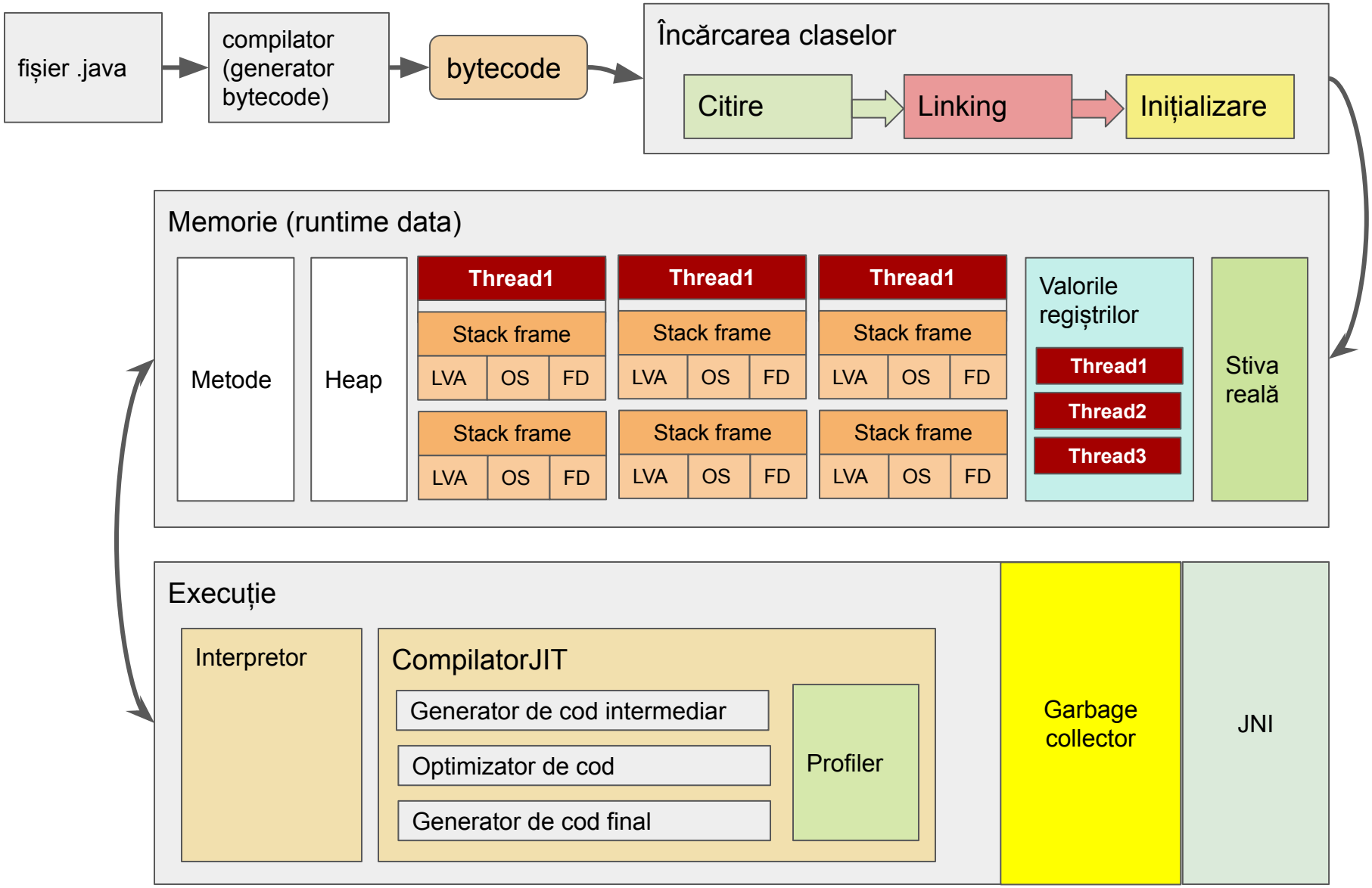
O mare parte din limbajele de programare noi compilează sursele generând ceea ce se numește "bytecode" - un fel de stream binar care are instrucțiuni pentru un procesor virtual, un CPU software emulat de o masina virtuala (process virtual machines, BCPL, cca 1966, apoi p-code - Pascal, 1970):

Java,Python,Php,Javascript,Ruby,Pascal,Visual Basic, etc.

Acestea sunt mașini virtuale "complete" - implementează un procesor virtual împreună cu un mic sistem de execuție virtual, asemănător unui sistem de operare care are propriul planificator de execuție, fire de execuție și management al memoriei.

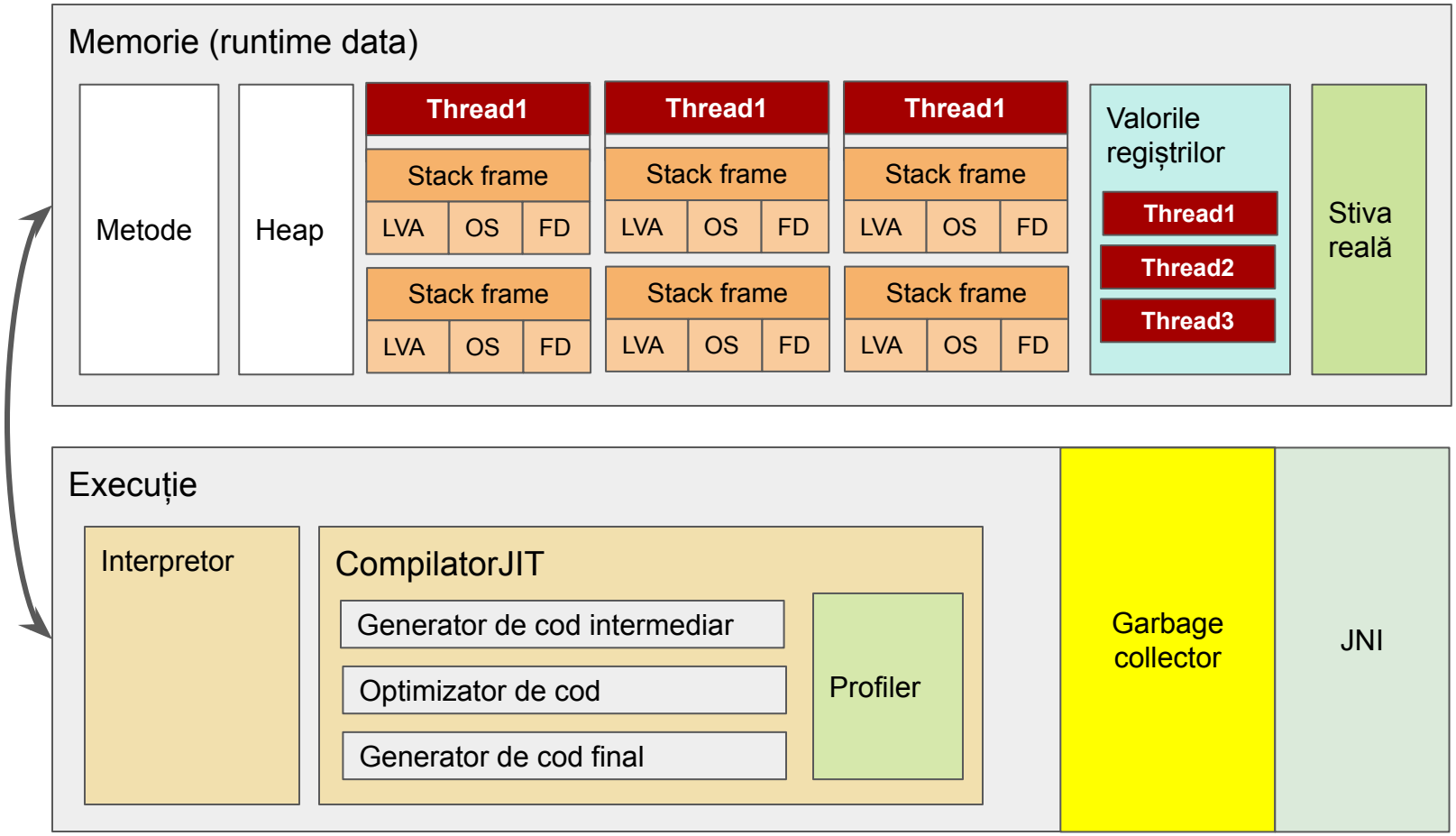
Mașinile virtuale permit detașarea rulării programului de suportul hardware (stack, regiștrii). Această detașare face relativ ușoară implementarea unor concepte care **extind paradigma programării procedurale**.

Process virtual machines (JAVA)



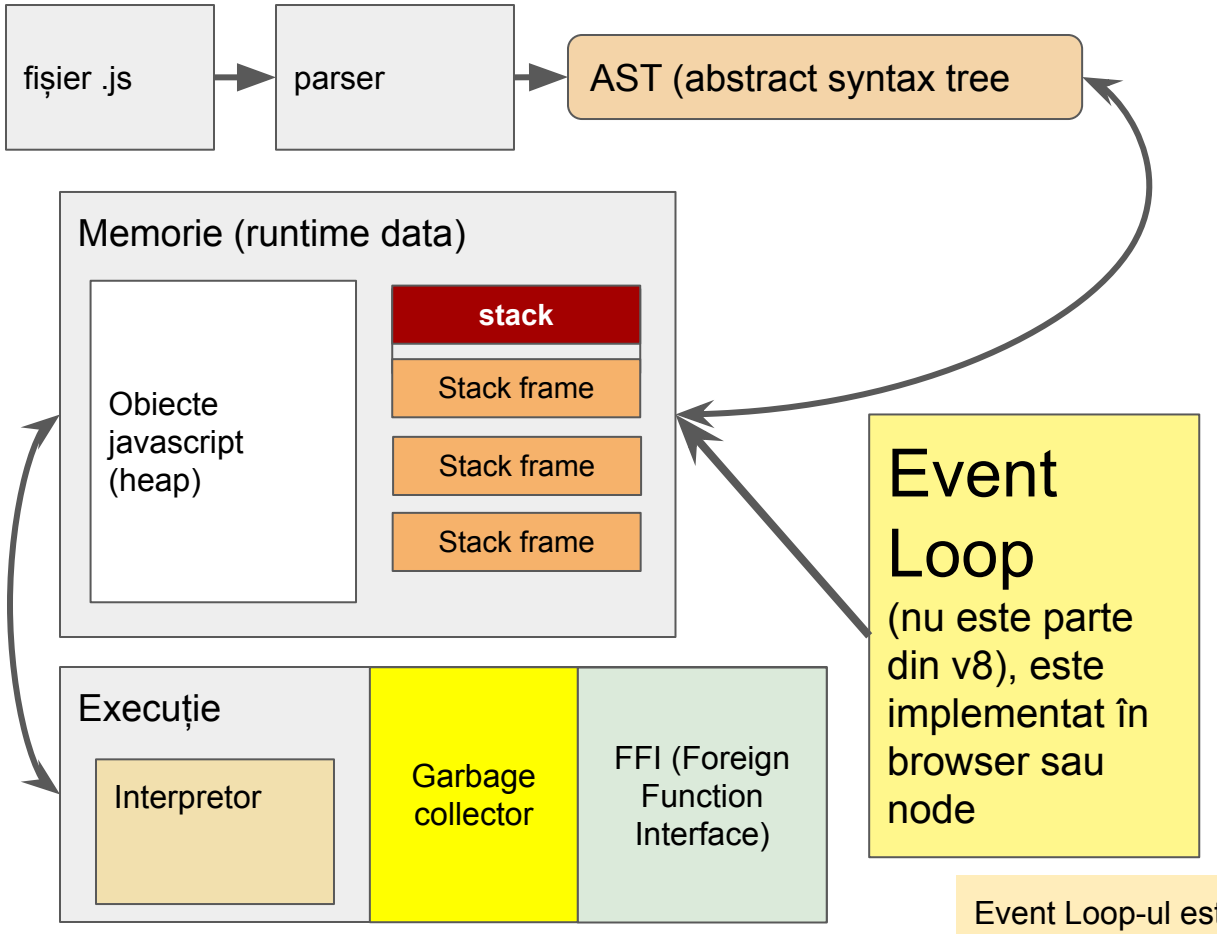
Process virtual machines (JAVA)

Structura datelor în memorie în mașina virtuală java este definită astfel încât sunt posibile mai multe threaduri, fiecare dintre ele având propria stivă și propriul set de valori pentru fiecare registru al procesorului. Threadurile JAVA sunt preemptive - sunt oprite de un manager, nu se așteaptă să cedeze controlul.



Bytecode-ul poate fi executat fie de interpretor (lent) fie de către JIT (Just In Time compiler) care compilează fragmente de cod transformându-le în cod nativ procesorului pe care se execută programul. Compilarea se face **în timpul execuției**, profilerul alegând fragmentele de cod care sunt recomandate spre compilare. JNI reprezintă un component care permite apelul funcțiilor externe (syscalls, de exemplu) cu pierdere a portabilitatii.

Process virtual machines (Javascript V8)



Javascript V8 este un interpretor care are **un singur thread**.

Execuția javascript nu este niciodată directă, ea este mediată de browsere sau de componente suplimentare gen node.js. Acestea adaugă alte componente.

Aceste componente suplimentare permit lansarea concomitentă mai multor threaduri V8

Aceste threaduri nu pot comunica între ele la fel de bine ca threadurile normale, de aceea este nevoie de **callback-uri** și de **event loop**

Event Loop se poate găsi și în Python, Php, Perl. Acesta este un component esențial al programării asincrone, orientate pe eveniment.

Event Loop-ul este un fel de scheduler de evenimente care execută operații suplimentare pe stack-ul v8.
Event Loop-ul este parte a browserului sau a node.js.
Apelurile AJAX (de exemplu) sunt făcute de browser, nu de V8.
Click-urile in browser sunt tot evenimente în Event Loop și au nevoie de callback-uri pentru a fi lansate în execuție la momentul potrivit.

Extinderi ale paradigmei procedurale

- Extinderile paradigmei procedurale sunt elemente care depășesc modelul execuției bazate pe existența unei singure stive în care funcțiile (procedurile) sunt reprezentate în timpul execuției prin frame-uri.
- Aproape toate aceste extinderi sunt inspirate din multitasking-ul implementat la nivelul sistemelor de operare. Aproape toate sunt implementate prin extinderea modului de interacțiune cu stiva principală de execuție. Aproape toate au ca scop creșterea eficienței unui program prin depășirea modelului secvențial în care orice perioadă de așteptare blochează complet execuția instrucțiunilor.
- Datorită necesității manipulării suplimentare a stivei de execuție, toate aceste extensii implică o combinație de extinderi ale compilatoarelor cu modalități de intervenție asupra execuției programului în timpul rulării. Acest lucru este mult mai simplu în limbajele interpretate care sunt executate în medii asemănătoare mașinilor virtuale.
- Limbajul C nu suportă în mod direct aceste extensii datorită relației foarte directe pe care o are cu hardware-ul care oferă doar modelul de execuție bazat pe o singură stivă cu intervenții foarte clare.
- Dar, aproape că nu există program modern care să nu utilizeze frecvent aceste extensii. De aceea, este important să clarificăm modul de implementare a acestora.

Extinderi ale paradigmei procedurale

- Toate conceptele prezentate în cele ce urmează sunt în realitate concepte de programare, despre care se știe ce se așteaptă de la ele și cum trebuie să funcționeze. Detaliile de implementare sunt de obicei mai puțin cunoscute și ele pot varia semnificativ de la un limbaj la altul.

- Dar, acestea sunt concepte de programare care NU POT Fi ÎNȚELESE doar pe baza modelului procedural clasic, bazat pe o singură stivă de execuție în care se construiesc frame-uri pentru fiecare funcție apelată.

- De aceea acestea sunt chiar extensii ale paradigmei - ale principiilor fundamentale ale modului procedural. Chiar dacă unele dintre ele au fost inventate în programarea funcțională (funcțiile anonime, callbackurile) este important faptul că acestea pot fi folosite acum într-un cod procedural.

- Ele se transferă și în paradigma obiectuală - toate conceptele prezentate în continuare pot fi utilizate și în programarea orientată obiect, cu modificările de sintaxă și implementare necesare.

- Callbacks
- Funcții anonime (sau lambda)
- Închideri (Closures)
- Generatori
- Corutine
- Asincronicitate

Callback

Un callback este o funcție care poate fi trimisă ca argument unei alte funcții.

callback.php

```php
<?php

doIt('myCallback');

function doIt($callback)
{
  $sir = "I find your lack of faith disturbing.";
  $callback($sir);
}

function myCallback($argument)
{
  print 'Am primit: ' .  $argument .  "\n";
}
```

https://onlinegdb.com/HC1SZsKqf

Funcția “doIt” primește ca argument numele unei alte funcții (myCallback). În interiorul funcției, variabila $callback poate fi executată cu transmiterea argumentului $sir.

Limbajul C suportă callbackuri cu ajutorul pointerilor către funcții, ajutat de faptul că sistemul de programare procedurală oricum stochează pointeri către adresele de intrare ale funcțiilor.

Funcția qsort, aflată în biblioteca stdlib.h necesită ca argument o funcție de comparare a două elemente.

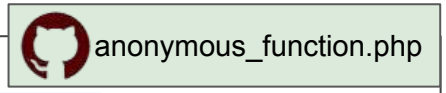
callback.c

```c
#include<stdio.h>

void callback(int n)
{
  printf("Eu sunt functia callback si am primit %d\n", n);
}

void executor(void (*ptr)())
{
  int mi = 13;
  (*ptr) (mi); // callback
}

int main()
{
  //se atribuie unui pointer o referinta catre functia callback
  void (*ptr)() = &callback;
  //se lanseaza in executie functia executor cu argumentul ptr
  executor(ptr);
  return 0;
}
```

https://onlinegdb.com/xpsoNWheN

Funcții anonime

Funcții (sau proceduri) care pot fi definite fără identificatori (fără nume specific de funcție).

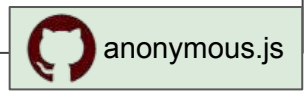



```php
<?php
$prn = function ($string)
{
  print "Sirul meu este: ". $string ."\n";
};

$prn("I find your lack of fate disturbing");
$prn("There's always a bigger fish");
```

https://onlinegdb.com/mEdU2bpkP

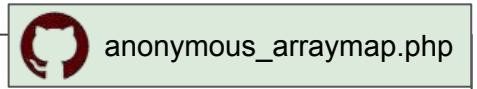
Funcția care determină afișarea șirurilor nu are nici un nume (este anonimă) dar poate fi asociată unei variabile și apelată ca atare.

```js
console.log('Incepem...')
setTimeout(function () {
  console.log('Executam dupa 1 secunda')
}, 1000);
```



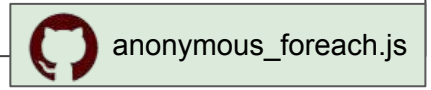
Javascript utilizează foarte frecvent funcții anonime, existând funcționalități care nu pot fi atinse altfel.

Funcțiile anonime sunt folosite frecvent pentru aplicarea unor prelucrări tuturor elementelor unui tablou.




```php
<?php
$nume = ["computer", "kompjuter",
"ordinador", "urdinatore", "tietokone"];

$entities = array_map(fn($nm) => print
$nm."\n", $nume);
```

```js
var nume = ["computer", "kompjuter",
"ordinador", "urdinatore", "tietokone"];

nume.forEach(function(nm, index) {
 console.log(index + " " + nm);
});
```



Limbajul de programare C nu permite funcții anonime. Anumite compilatoare adaugă extensii care permit aproximatii ale funcților anonime (gcc si clang de exemplu).

Închidere (Closure)

Funcții (proceduri) care pot fi definite în alte funcții și returnate, cu păstrarea parțială sau totală a frame-ului stivei.

Dacă putem asocia o funcție anonimă unei variabile înseamnă că putem să și returnăm o funcție din altă funcție?

closure.js

```
function topfunc() {
   var me = 'Sorin';
   var middlefunc = function(variabila) {
       print (variabila);  //print pentru rhino
       print (me);         //print pentru rhino
   }
   return middlefunc;
}

var mf = topfunc();
mf('ionel');
```

https://onlinegdb.com/lGv0jGILV

```
Ionel
Sorin
```

Variabila “me” este definită în interiorul funcției “topfunc”. Ea nu ar trebui să fie disponibilă în afara acestei funcții.

Cu toate acestea, la execuția funcției mf() care este returnată din funcția topfunc, variabila “me” pare a fi disponibilă pentru a fi printată. DE CE??

O funcție definită astfel, în limbajul javascript, se transformă în ceea ce se numește “closure” sau “Închidere”.

O ”Închidere” este o funcție anonimă care înregistrează și “contextul” acesteia - adică variabilele funcției în care ea a fost definită.

Astfel, funcția anonimă care este definită în funcția **middlefunc** va purta după ea și variabilele definite în funcția “middlefunc” (În acest caz, variabila “me”) și le va trata ca și cum ar fi fost definite în interiorul ei.

Închidere (Closure)

```php
<?php

function topfunc() {
  $me = 'Sorin';
  $middlefunc = function($var) {
     print ($var."\n");
     print ($me."\n");
  };
  return $middlefunc;
}

$mf = topfunc();
$mf('Ionel');
```

closure-error.php

https://onlinegdb.com/1Q-dNebNKL

```
Ionel
PHP Notice:  Undefined
variable: me in
/home/main.php on line 7
```

```php
<?php

function topfunc() {
  $me = 'Sorin';
  $middlefunc = function($var)  use ($me) {
     print ($var."\n");
     print ($me."\n");
  };
  return $middlefunc;
}

$mf = topfunc();
$mf('Ionel');
```

closure-corect.php

https://onlinegdb.com/rThTGWj8N

```
Ionel
Sorin
```

php cere specificarea variabilelor care să însoțească funcția de tip closure, spre deosebire de javascript care le păstrează pe toate automat.

Închidere (Closure)

Cum sunt implementate închiderile?

1. Compilatorul crează o structură de date în memorie care va conține o referință către codul executabil al funcției, împreună cu variabilele care trebuie stocate și valorile lor.
2. La execuția funcției, variabilele stocate în structura respectivă de date sunt copiate în stack-ul funcției ca variabile definite local.

Structura de date nu poate fi stocată în stack pentru ca frame-ul funcției părinte se va șterge după încheierea execuției acesteia și se vor pierde variabilei definite.

De aceea, această structură de date se definește în heap. Dar heap-ul nu are nici un mecanism de eliberare automată a memoriei în momentul în care alocarea ei nu mai este necesară. Ce se întâmplă cu aceste structuri de date după ce nu mai sunt necesare?

De obicei, limbajele care fac astfel de manipulări suplimentare a conținutului stackului trebuie să implementeze o posibilitate de a elibera automat acea memorie. Acest mecanism se numește “gunoier” (garbage collector).

Limbajul de programare C nu permite închideri. Gcc adaugă o extensie care permite definirea a ceea ce se numește “nested functions” . Acestea nu pot fi returnate, viața lor durează exclusiv în interiorul funcției părinte.

Generatori

Generatorii sunt funcții care pot ceda voluntar, temporar controlul funcției apelante.

generator.php

```php
<?php

function Generator()
{
  echo "computer","\n";
  yield;
  echo "kompjuter","\n";
  yield;
  echo "ordinador","\n";
  yield;
  echo "urdinatore","\n";
  yield;
}

$gen = Generator();

$value = $gen->current();
$gen->next();
$gen->next();
```

https://onlinegdb.com/2gyA0B6O-

```
php generator.php

computer
kompjuter
ordinador
```

generator1.php

```php
<?php

function Generator()
{
  yield "computer";
  echo "dupa primul yield\n";
  yield "kompjuter";
  echo "dupa al doilea yield\n";
  yield "ordinador";
  echo "dupa al treilea yield\n";
  yield "urdinatore";
  echo "dupa al patrulea yield\n";
}

$gen = Generator();
$value = $gen->current();
print $value."\n";
$gen->next();
$value = $gen->current();
print $value."\n";
$gen->next();
$value = $gen->current();
print $value."\n";
```

https://onlinegdb.com/KM2ngk1cn

```
php generator1.php

computer
dupa primul yield
kompjuter
dupa al doilea yield
ordinador
```

Generatorii pot fi și funcții anonime

generator2.php

```php
<?php

$mygen = function()
{
  yield "computer";
  echo "dupa primul yield\n";
  yield "kompjuter";
  echo "dupa al doilea yield\n";
  yield "ordinador";
  echo "dupa al treilea yield\n";
  yield "urdinatore";
  echo "dupa al patrulea yield\n";
};

$gen = $mygen();
$value = $gen->current();
print $value."\n";
$gen->next();
$value = $gen->current();
print $value."\n";
$gen->next();
$value = $gen->current();
print $value."\n";
```

https://onlinegdb.com/lehlxOvIj

```
php generator1.php

computer
dupa primul yield
kompjuter
dupa al doilea yield
ordinador
```

Generatori

generator1.php

```php
<?php

function Generator()
{
   yield "computer";
   echo "dupa primul yield\n";
   yield "kompjuter";
   echo "dupa al doilea yield\n";
   yield "ordinador";
   echo "dupa al treilea yield\n";
   yield "urdinatore";
   echo "dupa al patrulea yield\n";
}

$gen = Generator();

$value = $gen->current();
print $value."\n";
$gen->next();
$value = $gen->current();
print $value."\n";
$gen->next();
$value = $gen->current();
print $value."\n";
```

https://onlinegdb.com/KM2ngk1cn

Funcția Generator este apelată si începe să se execute. Execuția ei se suspendă (dar fără a se șterge stack frame-ul și fără returnare) la întâlnirea primei instrucțiuni yield. Yield poate returna sau nu o valoare.

La apelarea metodei “next”, execuția funcției Generator se reia din punctul în care a rămas, până la următorul yield. Toate informațiile asociate funcției (variabile locale, etc.) sunt păstrate. Funcția se execută până la următorul yield.

Metoda “next” se poate apela de câte ori este nevoie, până la epuizarea tuturor yield-urilor.

Limbajul C nu suporta direct generatori. Există o serie de extensii ale acestuia care pot adăuga această funcționalitate.

PseudoGeneratori

pseudogen.php

```php
<?php

$stare = 0;

function PseudoGenerator() {
   global $stare;
   if ($stare == 0) {
       $stare = 1;
       return "computer";
   } else if ($stare == 1) {
       $stare = 2;
       return "kompjuter";
   } else if ($stare == 2) {
       $stare = 3;
       return "urdinatore";
   } else if ($stare == 3) {
       $stare = 4;
       return "ordinador";
   }
}

$val = PseudoGenerator();
print "Val: ". $val ."\n";

$val = PseudoGenerator();
print "Val: ". $val ."\n";

$val = PseudoGenerator();
print "Val: ". $val ."\n";
```

https://onlinegdb.com/QZLPJEBuq

Generatori fără yield

Cheia implementării unui generator este menținerea informației de stare.

Aceasta se poate emula folosind o variabilă globală care poate fi accesibilă din interiorul funcției.

Un apel succesiv al funcției va returna valori diferite. Diferența este că funcția se va executa de fiecare dată de la început până la return-ul corespunzător.

În implementările reale, întreaga stivă funcției este înregistrată în heap (la apelarea instrucțiunii yield) și este reîncărcată în stiva principală la apelarea metodelor de tip next. Astfel, rămân disponibile la reluarea execuției toate variabilele locale definite în funcție.

Acest lucru este posibil în limbajele interpretate, executate în mașini virtuale pentru că acestea pot manipula stackul fără constrângerile impuse de hardware.

PseudoGeneratori în C

pseudogen.c

```c
#include <stdio.h>

int numara(void) {
   static int i, stare = 0;
   switch (stare) {
       case 0:
       for (i = 0; i < 10; i++) {
           stare = 1;
           return i;
           case 1:;
       }
   }
}

int main() {
   int c;
   c = numara();
   printf("c este: %d\n", c);
   c = numara();
   printf("c este: %d\n", c);
   c = numara();
   printf("c este: %d\n", c);
}
```

Variabilele care au clasa de stocare “static” sunt stocate în segmentul de date al memoriei și rămân vizibile pentru toate funcțiile fișierului curent. Astfel, pot fi utilizate pentru menținerea stării între apeluri succesive ale funcției.

Costrucția switch - case poate fi intercalată printre alte instrucțiuni astfel încât este posibil ca un case să fie în interiorul unei bucle for în timp ce switchul corespunzător se află în afara acestuia.

Atribuirea valorii 1 variabilei “stare” face ca la următoarea execuție a funcției construcția “case” să determine saltul direct în interiorul ciclului, după instrucțiunea “return”. Ciclul va sări la următorul i pe care îl va returna.

https://onlinegdb.com/xUIMXBk7U

Această utilizare a construcției “case” a fost probabil propusă de un angajat al companiei Lucasfilm numit “Tom Duff” care a folosit-o pentru optimizarea unei bucle de copiere. Optimizarea buclei este cunoscută sub numele “Duff’s device” (mai multe detalii: https://en.wikipedia.org/wiki/Duff%27s_device)

Corutine

Corutinele sunt asemănătoare subrutinelor, cu deosebirea că execuția lor poate fi suspendată și reluată de mai multe ori. Informațiile de stare ale unei corutine se mențin între rulări succesive.

Spre deosebire de generatori, corutinele pot primi parametri la rulările succesive.

Corutinele mențin propriul stack, în diverse moduri, în funcție de implementare. Astfel, corutinele sunt similare firelor de execuție (green threads) cu deosebirea că acestea cedează controlul voluntar, participând într-un mediu de multitasking cooperativ, spre deosebire de threads care sunt întrerupte de planificator (multitasking preemptiv).

coro1.py

```
def filter_line(ac):
    while True:
        line = (yield)
        if ac in line:
            print(line)


cor = filter_line("om")
next(cor)
cor.send("tietokone")
cor.send("computer")
cor.send("ordinateur")
cor.send("számítógép")
cor.send("ríomhaire")
```

https://onlinegdb.com/GGOaydknw

```
python coro1.py

computer
ríomhaire
```

Se definește funcția filter_line care este de tip corutina și care primește argumentul “ac”.

while True asigură execuția liniilor următoare de un număr infinit de ori.

Dacă este găsit șirul din variabila ac în interiorul șirului din variabila line, acesta este imprimat la consola.

Funcția filter_line este apelată, spre deosebire de apelarea unei funcții normale, aceasta NU este lansată în execuție.

Instrucțiunea next(cor) lansează funcția în execuție. Aceasta va fi executată până la prima instrucțiune (yield), unde este suspendată și începe să aștepte primirea unui argument

Se trimite parametrul “tietokone” care nu conține șirul “om”. Corutina va executa instrucțiunile până la următorul yield, unde se va opri si va aștepta.

Se trimite parametrul “computer” care nu conține șirul “om”. Corutina va executa instrucțiunile până la următorul yield, afișând la consolă șirul primit pentru că acesta conține “om”, apoi se va opri și va aștepta.

Corutine

```
def filter_line(num):
   while True:
       line = (yield)
       if num in line:
           yield(line)

cor = filter_line("om")
next(cor)
a = cor.send("tietokone")
if a: print(a)
a = cor.send("computer")
if a: print(a)
a = cor.send("ordinateur")
if a: print(a)
a = cor.send("számítógép")
if a: print(a)
a = cor.send("ríomhaire")
if a: print(a)
```

coro2.py

```
python coro2.py

computer
ríomhaire
```

https://onlinegdb.com/ryxHMkZw8u

Corutina poate returna parametrii, la fel ca generatorul. Aceasta se face de obicei tot cu ajutorul instrucțiunii yield. Compilatorul va detecta tipul de operație în funcție de sintaxă.

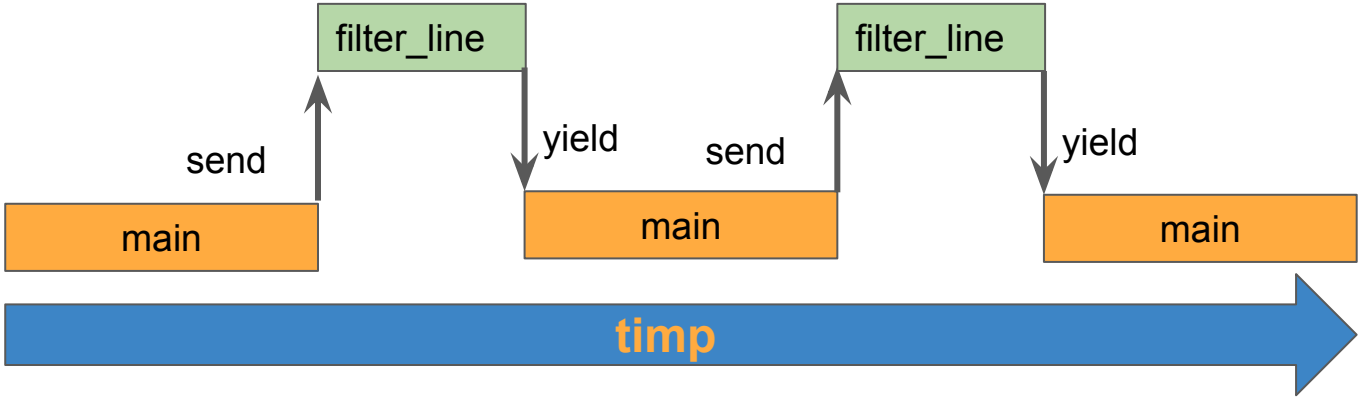
Limbajul C NU suportă corutine nativ datorită relației directe cu hardware-ul. Biblioteca standard C oferă un mecanism de salt necondiționat către o altă funcție, fără distrugerea stivei curente, folosind funcțiile setjmp și longjmp. Pe baza acestui mecanism, împreună cu funcțiile setcontext și getcontext, tot din biblioteca standard, au fost implementate mai multe tipuri de corutine.

Exemple:

https://code.google.com/archive/p/libconcurrency/
http://software.schmorp.de/pkg/libcoro.html
https://github.com/aolarchive/ribs2

Corutine - fire de execuție

Cele două funcții (main, funcția principală și filter_line, corutina apelată) nu rulează niciodată în același timp. Dar, ele există în același timp în memorie, fiecare dintre ele cu propria stivă.



Funcția “filter_line” cedează de fiecare dată voluntar controlul, în acest caz așteptând următorul parametru de la funcția apelantă. Cele două funcții se află într-o relație de multitasking cooperativ.

Corutina este un concept teoretic - acesta descrie un anumit comportament, nu o anumită implementare. De aceea, detaliile de implementare pot diferi semnificativ de la un limbaj de programare/compilator la altul. Anumite compilatoare au segmente speciale pentru implementarea corutinelor, altele se bazează pe threaduri sau fibre (în special fibre dacă sunt disponibile datorită multitaskingului cooperativ necesar).

Mecanismele descrise până acum se execută pe un singur procesor (nucleu)! Acestea reprezintă un mod de programare concurentă dar NU reprezintă în nici un fel paralelism.

Async - Await

Cu ajutorul multitaskingului cooperativ oferit de corutine și generatori, împreună cu closures și funcții anonime, este posibilă **o mai bună optimizare a proceselor de prelucrare a datelor** prin evitarea operațiilor care “blochează” procesorul degeaba. Majoritatea programelor de consum de date moderne folosesc asincronicitatea pentru a scădea timpul total de execuție, chiar în situația în care taskurile sunt executate pe același procesor.

Orice pagină WEB care solicită resurse adiționale textului propriu-zis (imagini, fișiere css, js, informații din alte surse, etc.) nu ar afișa absolut nimic tip de câteva secunde dacă nu ar permite afișarea asincronă a textului în timp ce se așteaptă încărcarea resurselor adiționale.

Din ce în ce mai multe limbaje de programare implementează **mecanisme de programare asincronă**. Cele mai frecvent utilizate cuvinte cheie în acest domeniu îl reprezintă perechea “**async/await**”.

De ce este importantă asincronicitatea?

Aproape toate programele care rulează pe un computer modern prelucrează cantități mari de date pe care trebuie să le obțină de la surse foarte lente. Harddiskul este o sursă lentă dar arhitecturile moderne implică obținerea datelor nu numai de la harddiskuri ci și de la surse externe aflate undeva pe internet.

O funcție care așteaptă date poate ceda controlul unei altei funcții urmând să-l primească înapoi atunci când începe să primească date. În acest mod se elimină timpii de așteptare.

În mod normal această treabă ar trebui făcută de către sistemul de operare dar pentru aceasta este nevoie de programare multiproces. Este dificilă, consumă foarte multe resurse (inițializarea proceselor durează). Astfel, se implementează o versiune de asincronicitate în interiorul unui singur program.

Async - Await - analogie

O maestră internațională la șah, foarte drăguță și extrem de bine îmbrăcată, își propune să joace un simultan cu 24 de jucători amatori.

Premise

- 24 de amatori
- Maestra are nevoie de 5 secunde pentru o mutare
- Fiecare amator are nevoie de 55 de secunde pentru o mutare
- Mutarile sunt făcute de o singura persoană (nici maestra nici amatorii nu ating tablele)
- Persoana care face mutarile se poate muta între table foarte repede (instantaneu)
- Jocurile au în medie 30 de perechi de mutări.

Varianta sincronă

Maestra joacă fiecare partidă separat, una câte una, fără să înceapă următoarea partidă până când partida curentă nu se termină.

Fiecare partidă durează 30 de minute *(55 + 5) * 30 = 1800 secunde*.

Întregul simultan durează 12 ore.

Varianta asincronă

Maestra se mută de la o tablă la alta, făcându-și mutarea lăsându-i pe amatori să se gândească tip de 55 de secunde.

O mutare a maestrei pentru toate cele 24 de jocuri durează 2 minute (*24 * 5 == 120* seconds), fiecare amator are la dispoziție cele 55 de secunde pentru a-și face mutarea.

Întregul simultan durează doar 1 oră (*120 * 30 == 3600* seconds).

Multitaskingul cooperativ NU paralelizează execuțiile, paralelizează asteptările.

Async - Await

https://onlinegdb.com/okFQpCjzO

```
import time

def show():
   print("kompjuter")
   time.sleep(1)
   print("ordinador")
   time.sleep(1)
   print("computer")

def main():
    show()
    show()
    show()
main()
```

sync-sleep.py

```
time python sync-sleep.py
kompjuter
ordinador
computer
kompjuter
ordinador
computer
kompjuter
ordinador
computer

real    0m6,040s
user    0m0,031s
sys     0m0,003s
```

https://onlinegdb.com/hlcIw12Wy

```
import asyncio

async def show():
   print("kompjuter")
   await asyncio.sleep(1)
   print("ordinador")
   await asyncio.sleep(1)
   print("computer")

async def main():
   await asyncio.gather(show(),
show(), show())

asyncio.run(main())
```

async-sleep.py

```
time python async-sleep.py
kompjuter
kompjuter
kompjuter
ordinador
ordinador
ordinador
computer
computer
computer

real    0m2,101s
user    0m0,088s
sys     0m0,008s
```

Se definește funcția asincronă “show”. De fapt, async transformă funcția show() într-o corutină.

await suspendă execuția corutinei si transfera controlul către funcția care a apelat-o.

Funcțiile asincrone trebuie apelate tot cu ajutorul funcției await.

Limbajul de programare C nu suportă programare asincronă. Există implementări care utilizează bibliotecile multithread ale sistemului de operare si exista posibilitatea asincronicitatii in I/O.

https://www.programiz.com/python-programming/online-compiler/

Async - Await

Tranziția de la programarea secvențială la programarea asincronă (concurentă) este probabil cea mai importantă modificare de paradigmă care s-a întâmplat în lumea programării în ultimii 10 ani.

Tranziția a fost determinată mai degrabă de nevoie decât de inovație - modelele concurente de execuție existau cu mult timp înainte de a apărea această creștere masivă a utilizării lor.

Această tranziție a făcut posibilă apariția unor noi arhitecturi software (ex: microservicii), noi produse (interfețele Web complexe) și a contribuit la creșterea vitezei majorității instrumentelor software cu care interacționăm frecvent.

Modul în care sunt implementate elementele de programare asincronă diferă de la o implementare a unui limbaj de programare la altul, multe limbaje având chiar mai multe modele. Acest mode de implementare se mai numește și "**modelul concurențial**" sau "**modelul de concurență**"

Toate implementările au în comun faptul că au nevoie de un component separat pentru managementul taskurilor asincrone. Acesta poate fi un event loop, un grup de task-uri sau threaduri, cu sau fără ajutorul sistemului de operare. Dar, de fiecare dată este nevoie de un instrument suplimentar pentru menținerea unui registru al activităților asincrone.

De exemplu, motorul de execuție Javascript folosit in Google Chrome (numit V8) nu permite asincronicitate. Aceasta este adăugată ca parte a browserului. Node.js, care este un motor de execuție a programelor javascript pe server, deși folosește tot v8 are un alt component care asigură asincronicitatea. Chrome folosește https://libevent.org/ iar node.js folosește https://libuv.org/. Ambele se folosesc de apeluri de sistem al sistemului de operare.

v8 nu contine instrucțiunea setTimeout()!! Ea, împreună cu altele similare sunt implementate în event loop.

Viitorul programării procedurale

Conceptele de callback, funcții anonime, closure, corutine, generatori reprezintă extensii ale paradigmei procedurale, disponibile în special (dar nu numai) în limbajele executate în mașini virtuale.

Cu toate acestea, toate aceste instrumente nu sunt limitate la paradigma programării structurate. Ele sunt disponibile și în programarea orientată obiect și în programarea funcțională.

Este important de reținut că accepțiunea modernă a termenului “programare structurată” include aceste instrumente. De aceea există nenumărate voci care afirmă că viitorul programării (cel apropiat) va fi marcat de o revenire la programarea structurată, evitându-se încapsularea - care este extrem de costisitoare în cazul fluxurilor mari de date.

Programarea orientată obiect presupune instanțierea unei ierarhii de obiecte pentru reprezentarea datelor. Se urmărește mai degrabă corectitudinea arhitecturii, nu eficiența resurselor consumate.

De exemplu, reprezentarea unui rând dintr-un tabel aflat într-o bază de date într-un sistem obiectual conduce la o cantitate de 1.2Mocteți de date serializate prin comparație cu dimensiunea reală a datelor care este de 412 octeți.

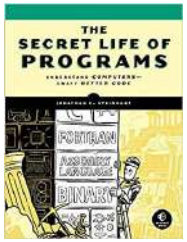
Exemplul este extrem, serializarea adaugă extrem de mult la memoria reală ocupată dar contrastul este evident. Dacă avem de procesat 100 - 1000 de rânduri este acceptabil dar dacă avem de procesat milioane de rânduri nu mai este.

11. Materiale suplimentare recomandate

Orice ocupație profesională este compusă, pe de-o parte, dintr-un set de abilități pe care cineva trebuie să le deprindă, iar pe de altă parte, dintr-o cunoaștere acceptabilă a culturii ocupației respective. Abilitățile se pot antrena dar ele de obicei sunt ca niște instrumente precise, care se pot aplica abia după ce este recunoscută și definită problema care trebuie rezolvată. Recunoașterea și definirea problemei se face de cele mai multe ori pe baza unei intuiții aparte, intuiție care se dezvoltă prin cunoașterea culturii domeniului respectiv.

Materialele care urmează se constituie într-o listă eterogenă de interviuri cu inventatorii unor limbaje de programare, prezentări ale unor concepte discutate în detaliu, comentarii despre starea domeniului programării și multe altele. Majoritatea sunt materiale video, însoțite de câteva cărti. Materialele video sunt însoțite de scurte descrieri.

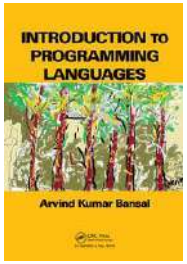
WOULD YOU LIKE TO KNOW MORE?



Jonathan E. Steinhart - **The Secret Life of Programs**
Understand Computers -- Craft better code

No Starch Press, Inc., 2019

ISBN: 978-1593279707



Arvind Kumar Bansal -
Introduction to Programming Languages

Chapman and Hall, 2013

ISBN: 978-0429100338

4 Programming Paradigms In 40 Minutes

O prezentare a principalelor paradigme de programare, cu exemple foarte simple de cod (utilă pentru cei care nu au cunoștințe de programare).

https://www.youtube.com/watch?v=cgVVZMfLjEI

Programming Across Paradigms

O prezentare a principalelor paradigme de programare, cu exemple de cod (utilă pentru cei care au cunoștințe de programare). 49.07 min.

https://www.youtube.com/watch?v=Pg3UeB-5FdA

Building an 8-bit breadboard computer!

Pentru cei interesați de arhitectura computerelor - un playlist care detaliază construcția unui computer pe 8 biti pe o placă de tip breadboard cu fire și leduri. 44 de clipuri a câte 30 min fiecare, în medie.

https://www.youtube.com/playlist?list=PLowKtXNTBypGqImE405J2565dvjafglHU

WOULD YOU LIKE TO KNOW MORE?

Pradip Dey, Manas Ghosh - **Computer Fundamentals and Programming in C**

Oxford University Press, 2013

ISBN-13: 978-0198084563

Brian Kernighan, Dennis Ritchie - C Programming Language, editia 2

Prentice Hall, 1988

ISBN: 978-0131103627

Uncle Bob Martin - the future of programming

Robert Martin (“unchiul Bob”) este un teoretician si practician al principiilor de dezvoltare software, fiind inițiatorul principiilor SOLID si a sistemului AGILE de management de proiect.

https://www.youtube.com/watch?v=ecIWPzGEbFc

History and Spirit of C - Olve Maudal

O prezentare a istoriei dezvoltării programării, ajungând la apariția limbajului C

https://www.youtube.com/watch?v=xGVRF-Y--hI

Brian Kernighan: UNIX, C, AWK, AMPL, and Go Programming | Lex Fridman Podcast

Interviu cu Brian Kernighan - coautor al primului manual de C, participant la proiectarea sistemului UNIX, etc.

https://www.youtube.com/watch?v=O9upVbGSBFo

WOULD YOU LIKE TO KNOW MORE?

Floating Point Numbers (Part1: Fp vs Fixed) - Computerphile

Cât de mult crește viteza calculelor prin utilizarea unui procesor de numere flotante?

https://www.youtube.com/watch?v=f4ekifyijIg

Fixed Point Arithmetic 1: Intro to Fixed Point

Detalii despre modalitatea de a stoca numerele fracționare cu ajutorul virgulei fixe

https://www.youtube.com/watch?v=S12qx1DwjVk

Where did Bytes Come From? - Computerphile

Despre originea conceptului de octet.

https://www.youtube.com/watch?v=ixJCo0cyAuA

WOULD YOU LIKE TO KNOW MORE?

Comparing C to machine language

O Comparație a unui program simplu în limbajul C (calculul numerelor Fibonacci) cu codul generat în limbajul de asamblare.

https://www.youtube.com/watch?v=yOyaJXpAYZQ

The Art of Code - Dylan Beattie

Despre utilizarea computerelor pentru a crea arta - chiar dacă scopul inițial nu este unul artistic.

https://www.youtube.com/watch?v=uQXXMycJwlM

5. C to Assembly - MIT OpenCourseWare

Mult mai multe despre modul in care este implementat limbajul C în assemblerul procesoarelor x86 - 64.

https://www.youtube.com/watch?v=wt7a5BOztuM

WOULD YOU LIKE TO KNOW MORE?

What the heck is the event loop anyway? | Philip Roberts | JSConf EU

O prezentare a implementării event-loop în javascript și a modului în care funcționează

https://www.youtube.com/watch?v=8aGhZQkoFbQ

The Forgotten Art of Structured Programming - Kevlin Henney [C++ on Sea 2019]

O discuție despre relevanța modelului de programare structurată în programarea modernă

https://www.youtube.com/watch?v=SFv8Wm2HdNM

Object-Oriented Programming is Bad

Motivele pentru care în multe situații programarea structurată (procedurală) este preferabilă unei încercări de a proiecta o arhitectură orientată obiect.

https://www.youtube.com/watch?v=QM1iUe6IofM

I programmed some creatures. They Evolved.

Prezentarea unui model de evolutie a populatiei in care se vede cum indivizi cu structură extrem de simplă conduc la comportamente sociale complexe prin evoluție

https://www.youtube.com/watch?v=N3tRFayqVtk

WOULD YOU LIKE TO KNOW MORE?

21:Everything u need 2 know about pointers -Richard Buckland

Pointeri, referințe - o lecție la tablă

https://www.youtube.com/watch?v=Rxvv9krECNw

5. C to Assembly

Detalii despre cum este tradus codul C în limbajul de asamblare. Se vorbeste si despre LLVM - un compilator modern care poate compila mai multe limbaje de programare

https://www.youtube.com/watch?v=wt7a5BOztuM

The Next Decade of Software Development - Richard Campbell - NDC Oslo 2021

O incercare de a imagina viitorul imediat al dezvoltarii software pe baza celor intamplate pana acum, împreună cu ceva sfaturi cu privire la modul în care ar trebui abordată o carieră în programare.

https://www.youtube.com/watch?v=a_LM3FJozr0

WOULD YOU LIKE TO KNOW MORE?

Plain Text - Dylan Beattie - NDC Oslo 2021

Reprezentarea textului - Morse, ASCII, UTF, etc., erori celebre determinate de codificarea textului și de ce are Billy Joel un album numit "Kohuept"

https://www.youtube.com/watch?v=_mZBa3sqTrI

Unicode and Byte Order

Reprezentarea textului - despre diferitele tipuri de codificare a caracterelor și influența ordinii de stocare a octeților.

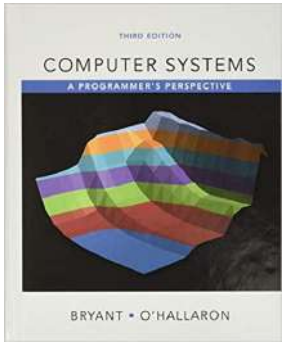
https://www.youtube.com/watch?v=bbkUn0o3L1Y

David Patterson: Computer Architecture and Data Storage | Lex Fridman Podcast #104

David Patterson este unul dintre inventatorii arhitecturilor de tip RISC (probabil cel care a inventat termenul) și al arhitecturii de stocare de tip RAID.

https://www.youtube.com/watch?v=naed4C4hfAg

WOULD YOU LIKE TO KNOW MORE?



Computer Systems: A Programmer's Perspective

Randal Bryant, David O'Hallaron

ISBN: 978-0134092669

GitHub - shihyu/CSAPP2e

Programming Paradigms (Stanford)

Cursul de programare de la Stanford

https://www.youtube.com/watch?v=Ps8jOj7diA0&list=PLD28639E2FFC4B86A

000000000