Fundamentele Programarii

Tipuri de date definite de utilizator

Tipuri de date definite de utilizator

- Redenumiri de tipuri
- ✓ Enumerari (o lista de valori simbolice care sunt mapate la valori intregi)
- Structuri
- Uniuni

Redenumiri de tipuri

Limbajul C/C++ dă posibilitatea programatorului să introducă denumiri personalizate pentru tipurile limbajului, cu ajutorul cuvântului cheie ***typedef***

Redenumirea unui anumit tip este realizată astfel: scriem declaraţia fără iniţializare a unei date de tipul vizat şi apoi transformăm această declaraţie de dată într-o *declaraţie de tip* scriind cuvântul cheie ***typedef*** în faţa sa. În acest fel, numele datei devine numele tipului redenumit. Astfel, instrucţiunea

```
int Intreg;
```

declară variabila ***Intreg*** de tip ***int***, iar instrucţiunea

```
typedef int Intreg;
```

declară tipul ***Intreg*** ca o redenumire a tipului ***int***.

Prin convenţie, denumirile de tipuri **se scriu cu majusculă pe primul loc**, sau **numai cu majuscule**, sau **numai cu litere mici dar cu sufixul “ *_t*”** :

```
typedef unsigned Natural, UINT, size_t;
```

Redenumiri de tipuri

Redenumirile uneori sunt utile în cazul tipurilor compuse:

```
#include<iostream>
using namespace std;
typedef int Tablou[5], Intreg;
typedef Tablou Matrice[2];

int main(){
        Tablou tab={1,2,3,4,5};
        for(Intreg i=0;i<5;i++)
                cout<<tab[i]<<endl;
        Matrice mat={{10,20,30,40,50}, {11,12,13,14,15}};
        for(Intreg i=0;i<2;i++){
                for(int j=0;j<5;j++)
                        cout<<mat[i][j]<<' ';
                cout<<endl;
                }
        return 0;
}
```

Redenumari de tipuri

Atentie! O declaraţie ***typedef*** nu defineşte un tip nou de dată, ea introduce numai o nouă denumire pentru un tip deja existent.

Programatorul are totuşi posibilitatea să definească şi tipuri noi de date, dar numai dacă acestea se încadrează în una din următoarele patru categorii:

- *enumerări*,
- *structuri*,
- *uniuni*,
- *clase* (vor fi studiate la POO) .

- Aceste patru tipuri sunt numite *tipuri utilizator*, deoarece programatorul este considerat ca fiind un utilizator al limbajului.
- Aceste noi tipuri utilizator pot modela date si concepte din lumea reala!

Structuri

- *O structură* (*structure*) este un ansamblu format din una sau mai multe *variabile (câmpuri)* grupate împreună sub un singur nume.

- Datele de tip structură au pătruns în limbajele de programare în primul rând pentru facilitarea manipulării datelor de gestiune economică.

- Utilizarea structurilor s-a extins considerabil odată cu dotarea lor, pe lângă membrii de tip variabilă (*câmpuri*), cu membri de tip funcţie (*metode*). Această extindere a făcut trecerea de la C la C++ şi, pentru a marca schimbarea majoră de viziune asupra structurilor, ele au fost redenumite *clase*.

- În C++ nu sunt diferenţe esenţiale între structuri şi clase.

- Vom prezenta structurile aşa cum au fost ele implementate iniţial în limbajul C.

Structuri

Structurile se deosebesc de tablouri prin următoarele aspecte:

I. elementele membre ale unei structuri pot avea **tipuri diferite**;

II. structurile se comportă la alocare exact ca variabilele simple: dacă sunt locale sunt alocate pe stivă, altfel sunt alocate în zona variabilelor globale, **funcţiile pot returna structuri**;

III. **elementele** unui structuri nu sunt variabile anonime ci **au câte un nume**;

IV. referirea unui element al unei structuri se realizează cu operatorul de selecţie (**.**) şi nu cu operatorul de indexare ([ ]).

Structuri

Structurile sunt definite cu ajutorul cuvantului cheie ***struct***, iar o declaraţie de structură introduce un nou tip de dată: cel tocmai precizat. Exemplu, **declaraţia**

```
struct Complex{
   double x;
   double y;
    };
```

defineşte tipul ***Complex***, fiecare dată de tip ***Complex*** fiind un *obiect* de tip structură compus din două câmpuri membre, ***x*** şi ***y***, ambele tip ***double***. O variabilă de acest tip se declară, în C, astfel:

```
struct Complex w;
```

În C++ nu mai este obligatorie utilizarea cuvântului cheie ***struct*** la declararea unei variabile (dacă structura a fost deja definită), deci declaraţia de mai sus poate fi scrisă astfel:

```
Complex w;
```

Accesul la câmpurile membre este dat de operatorul de selecţie "punct":

```
cout<<w.x<<endl;
```

Structuri

Iniţializarea unei variabile de tip structură poate fi facută odată cu declararea ei, respectând strict ordinea membrilor din definiţia structurii:

```
Complex w={1.5, 3.0};
```

sau dupa declarare:

```
Complex w;
w={1.5, 3.0}; // w.x=1.5; w.y=3.0;
```

De asemenea, variabilele pot fi declarate odată cu definiţia structurii, astfel:

```
struct Complex{
    double x;
    double y;
} z1,z2, z3={1.5, 3.3};
```

Pot fi declarate şi structuri anonime, caz în care toate variabilele de acest tip trebuie declarate (nu neapărat și inițializate) de la început:

```
struct {
    double x;
    double y;
} z1,z2,z3={1.5, 3.3};
```

Vezi Programul Structuri

Uniuni

O dată de tip ***uniune*** este o structură "colapsată“ (comprimată): toţi membrii ei sunt suprapuşi în acelaşi spaţiu de memorie, pentru că nu există spatiu pentru fiecare membru individual.

O uniune se declară exact la fel ca o structură, schimbând doar cuvântul cheie ***`struct`*** cu ***`union`*** dar, spre deosebire de cazul unei structuri, unei **uniuni nu i se alocă decât spaţiul de memorie strict necesar pentru a cuprinde cel mai expansiv membru (cel care necesita cel mai mult spatiu de memorie)**, toţi membrii uniunii urmând sa fie alocaţi, rând pe rând, în această zonă comună.

În consecinţă, membrii unei uniuni nu pot fi folosiţi simultan ci numai succesiv: ultimul alocat este utilizat în mod valid până la alocarea altuia. **Programul trebuie să ştie permanent care membru al uniunii are reprezentarea corectă în memorie în acel moment.** Uniunile sunt utilizate pentru economisirea memoriei alocate programului.

Vezi Programul Uniuni

https://test-master.space

Fundamentele Programarii

Functii

Functii

- Prototip
- Definitie
- Parametrii
- Instructiunea `return`

- Apelul parametrilor prin valoare
- Supraincarcarea functiilor
- Parametrii functiilor- tablouri

- Apelul parametrilor prin referinta
- Vizibilitatea entitatilor
- Cum functioneaza apelarea unei functii
- Functii inline
- Functii recursive

Ce este o functie?

- O **funcție** este un bloc de cod care execută o anumită sarcină, organizată într-o structură definită, ce poate fi apelată ori de câte ori este necesar în program. Funcțiile ajută la structurarea codului, la evitarea redundanței și la creșterea lizibilității și reutilizabilității.

- In programele C++ folosim o serie de functii (si clase) care sunt definite in Librariile Standard C++
- Exista si alte librarii care pot fi folosite
- Putem face si functiile si clasele noastre

- Functiile permit **modularizarea** unui program (divizarea programului in subprograme mai mici)
- Functiile permit structurarea programelor în secvențe mai mici de cod care indeplinesc anumite sarcini si care pot fi indepedente.
- Aceste secvente pot fi folosite de mai multe ori

Ce este o functie?

```
int main(){

    //citire
    Instructiune1;
    Instructiune2;
    Instructiune3;

    //procesul de prelucrare a datelor
    Instructiune4;
    Instructiune5;
    Instructiune6;

    //afisare
    Instructiune7;
    Instructiune8;
    Instructiune9;

    return 0;
}
```

Cod modularizat

```
int main(){

    //citire
    functie_citire();

    //procesul de prelucrare a datelor
    functie_prelucrare();

    //afisare
    functie_afisare();

    return 0;
}
```

Ce este o functie?

```
functie_citire(){
	Instructiune1;
	Instructiune2;
	Instructiune3;
}

functie_prelucrare(){
	Instructiune4;
	Instructiune5;
	Instructiune6;
}

functie_afisare(){
	Instructiune7;
	Instructiune8;
	Instructiune9;
}
```

```
int main(){

	functie_citire();

	functie_prelucrare();

	functie_afisare();

	return 0;
}
```

Ce este o functie?

- Analogia dintre **functia main/celelalte functii** cu **seful/angajatii**

Seful trebuie sa rezolve o anumita problema mai ampla, care poate fi impartita in mai multe probleme mai mici.
Rolul sefului nu este sa rezolve el toate problemele care apar ci

- sa coordoneze executia si rezolvarea acestor probleme,
- sa sintetizeze rezultatele si sa le prezinte propriului sau sef.

Seful angajeaza 3 experti care sa lucreze la cate una din problemele mai mici si le *ofera toate informatiile* de care au nevoie pentru a-si duce fiecare din ei treaba la bun sfarsit.
Cum fiecare angajat stie foarte bine ce are de facut, dupa ce rezolva problema care i-a fost atribuita, raporteaza indeplinirea cerintei sefului. Raportul poate fi unul scurt "Am rezolvat" sau poate fi unul mai complex, in care se ofera mai multe informatii.
Seful primeste toate rapoartele, le sintetizeaza, apoi le trimite mai departe propriului sau sef.

Pare usor sa fii in pozitia sefului?

- trebuie sa stie precis ce sarcina sa ii atribuie fiecarui angajat,
- trebuie sa stie ordinea in care sintetizeaza raspunsurile primite,
- trebuie sa ofere toate informatiiile necesare angajatilor pentru ca acestia sa poate rezolva problema care i-a fost data
- trebuie sa stie ce fel de raport va primi inapoi de la fiecare anagajat pentru a stii cum sa faca sinteza si prezentarea solutiei problemei mai departe

Ce este o functie?

Functia **`main():`**

- Trebuie sa stie sa apeleze functiile pe care le foloseste
- Trebuie sa stie care este utilitatea fiecarei functiei utilizate
- Trebuie sa stie si sa inteleaga ce informatii sunt necesare pentru a crea acele functii
- Trebuie sa inteleaga ce va returna fiecare functie
- Trebuie sa inteleaga ce erori poate produce fiecare functie
- Trebuie sa inteleaga daca avem sau nu anumite constrangeri de performanta

- Daca folosim functii din librariile standard nu ne va interesa in mod special cum functioneaza ele intern, Important e sa stim sa le folosim corect!

Ce este o functie?

- Exemplu `<math.h>`

- Calcule matematice uzuale
- Functii globale

```
Nume_functie (argument);
Nume_functie (argument1, argument2,….);
```

```
cout<<sqrt(400.0)<<endl;                    //20.0
double rezultat;
rezultat=pow(2.0,3.0);                      //2.0^3.0
```

- **Abstractizare**= procesul care ofera doar detaliile esentiale, ascunzand detaliile implemantarii

- Exemplu: noi folosim ceasul pentru a vedea cat este ora, dar nu stim mecanismul din spate, cum functioneaza pentru a afisa ora.

Ce este o functie?

- Pentru a vedea ce functii sunt in librariile standard C++: https://en.cppreference.com/w/cpp/header
- Vezi programul “CMath”!

Ce este o functie?

Functii definite de programator

- Putem defini propriile noastre functii
- **Exemplu:** Vrem sa cream o functie care sa adune doua numere intregi

```
int suma_numere (int a, int b){

	return a+b;
}

int main(){
	cout<<suma_numere(20,40);
	return 0;
}
```

Ce este o functie?

- **Exemplu:** Vrem sa cream o functie care sa adune doua numere pozitive, si sa returneze 0 in cazul in care cel putin unul dintre numere este negativ

```
int suma_numere (int a, int b){

	if(a<0 || b<0)
		return 0;
	else
		return a+b;
}

int main(){
	cout<<suma_numere(20,40)<<endl;		//60
	cout<<suma_numere(-20,40)<<endl;	//0
	return 0;
}
```

Cum definim o functie?

Pentru a defini o functie trebuie sa avem in vedere urmatoarele:

- **Numele functiei**
  - respecta aceleasi reguli ca si numele variabilelor
  - trebuie sa fie potrivit cu ceea ce dorim sa facem in cadrul acelei functii
  - nici prea lung, nici prea scurt

- **Lista parametrilor**
  - variabilele pe care trebuie sa le dam atunci cand apelam functia
  - tipul lor trebuie specificat
  - e posibil ca functia sa nu aiba parametrii

- **Tipul returnat**
  - tipul datei pe care functia o returneaza: bool, int, double, string, struct, enum
  - e posibil ca functia sa nu returneze nimic si atunci spunem ca tipul functiei este *void*

- **Corpul functiei**
  - instructiunile care vor fi executate atunci cand apelam functia
  - se scrie intre acolade { }

Cum definim o functie?

Exemplu de functie fara parametrii:

```
int nume_functie ( ) {
        instructiuni;
        return 0;
}
```

1.Nume
2. Parametrii
3. Tipul returnat
4. Corpul functiei

- O astfel de functie este functia *main ( )*

Exemplu de functie cu 1 parametru:

```
int nume_functie ( int a) {
        instructiuni;
        return 0;
}
```

1.Nume
2. Parametrii
3. Tipul returnat
4. Corpul functiei

Cum definim o functie?

Exemplu de functie care nu returneaza nimic (void)

```
void nume_functie ( ) {
        instructiuni;
        return; //optional
}
```

1.Nume
2. Parametrii
3. Tipul returnat
4. Corpul functiei

Exemplu de functie cu mai multi parametrii:

```
void nume_functie ( int a, string b) {
        instructiuni;
        return ; //optional
}
```

1.Nume
2. Parametrii
3. Tipul returnat
4. Corpul functiei

- Atentie, cand vom apela functia trebuie sa dam parametrii exact in ordinea in care apar atunci cand declaram functia!

Cum definim o functie?

Exemplu de functie care nu returneaza nimic (void) si nu are nici parametrii

```
void say_hello ( ) {
        cout << “Hello” << endl;
}

int main ( ) {
        say_hello();          //Hello
        return 0;
}
```

1.Nume
2. Parametrii
3. Tipul returnat
4. Corpul functiei

Cum apelam o functie?

```
void say_hello ( ) {
        cout << “Hello” << endl;
}

int main ( ) {
        for(int i {1}; i<= 10; i++)
                say_hello();
        return 0;
}
```

```
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

Cum apelam o functie?

- Putem avea oricate functii dorim intr-un program

```
void say_world ( ) {
        cout << “World!” << endl;
}

void say_hello ( ) {
        cout << “Hello” <<“  ”;
        say_world ( );
}

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

Hello World!

Cum apelam o functie?

- Putem avea oricate functii dorim intr-un program

```
void say_world ( ) {
        cout << “World!” << endl;
        cout << “Bye from say_world” << endl;
}

void say_hello ( ) {
        cout << “Hello” <<“  ”;
        say_world ( );
        cout<<“Bye from say_hello”<<endl;
}

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

Hello World!
Bye from say_world
Bye from say_hello

Cum apelam o functie?

- Functiile pot apela alte functii
- Compilatorul trebuie sa cunoasca toate informatiile despre functie **inainte** de a o apela

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

void say_hello ( ){
        cout << "Hello" <<"  ";
}
```

Eroare

- Vezi programul "AriaCercVolumCilindru"!

Prototipul functiilor

- In urma cu cateva slide-uri am mentionat faptul ca atunci cand compilatorul foloseste o functie trebuie sa cunosca deja toate informatiile despre acea functie. Pentru a respecta acest lucru putem proceda in doua moduri:

  - Sa defim functiile inainte de a le apela
    - Ok pentru programe mici
    - Nu e o solutie practica pentru programe mai mari

  - Sa folosim prototipul functiilor
    - Spunem compilatorului tot ce are nevoie sa cunoasca fara a da intreaga definitie a functiei
    - Se scrie la inceputul programului
    - Se pot scrie si intr-un fisier header (.h)

Prototipul functiilor

Exemple:

- Functie fara parametrii

```
int nume_functie ( );     // prototipul


int nume_functie( ){
        instructiuni;
        return 0;
}
```

- Functie cu parametrii

```
int nume_functie ( int );     // prototipul
//sau
int nume_functie ( int a );     // prototipul

int nume_functie( int a ){
        instructiuni;
        return 0;
}
```

Prototipul functiilor

Exemple:

- Functie care nu returneaza nimic, fara parametrii

```
void nume_functie ( );
// prototipul


void nume_functie( ){
        instructiuni;
        return; //optional
}
```

- Functie care nu returneaza nimic, cu parametrii

```
void nume_functie ( int a, string b );
// prototipul sau

void nume_functie ( int, string );
// prototipul

void nume_functie( int a, string b ){
        instructiuni;
        return; //optional
}
```

Prototipul functiilor

Exemple:

```
void say_hello( );

int main ( ) {
        say_hello ( );          //OK
        say_hello(100);       //eroare
        cout<<say_hello ( );    //eroare
        return 0;
}

void say_hello ( ) {
        cout << “Hello” <<“  ”;
}
```

Prototipul functiilor

Exemple:

```
void say_hello( );
void say_world( );

int main ( ){
        say_hello ( );
        cout<<"Bye from main"<<endl;
        return 0;
}

void say_hello ( ){
        cout << "Hello" <<"  ";
        say_world ( );
        cout<<"Bye from say_hello"<<endl;
}

void say_world ( ){
        cout << "World!" << endl;
        cout << "Bye from say_world" << endl;
}
```

```
Hello World!
Bye from say_world
Bye from say_hello
Bye from main
```

Vezi programele "Prototip", "Prototip-header"!

Parametrii functiilor

- Atunci cand apelam o functie putem sa ii transmitem niste date functiei respective
- Atunci cand apelam functia aceste date se numesc **argumente**
- In definitia functiei aceste date se numesc **parametrii**
- Trebuie sa fim atenti ca argumentele pe care le dam atunci cand apelam functia sa se potriveasca cu parametrii functiei pe care i-am dat atunci cand am definit functia
(numarul lor sa fie acelasi, ordinea in care apar sa fie aceeasi, tipul lor sa fie acelasi)

Parametrii functiilor

- **Exemplu**

```
int suma_numerelor (int, int);      //prototipul

int main ( ) {
        int rezultat { };
        rezultat = suma_numerelor (100, 200);    //apelarea functiei
        return 0;
}

int suma_numerelor (int a, int b){          //definitia functiei
        return a+b;
}
```

Parametrii functiilor

- **Exemplu**

```
int diferenta_numerelor (int, int);      //prototipul

int main ( ) {
        int rezultat { };
        rezultat = diferenta_numerelor (100, 200);    //apelarea functiei
        rezultat = diferenta_numerelor (200, 100);
        return 0;
}

int diferenta_numerelor (int a, int b){          //definitia functiei
        return a-b;
}
```

Parametrii functiilor

- **Exemplu**

```
void say_hello (string name){   //parametrul este un obiect de tip string C++
        cout << "Hello " << name << endl;
}

int main ( ) {
        say_hello("Simona");   //argumentul este un string literal C (compilatorul il converteste
                               //la un string C++)
        string caine {"Horia"};
        say_hello (caine);
        return 0;
}
```

Apelul functiilor

- Apelul unei functii poate fi realizat prin **valoare** sau prin **referinta**.

Apelul prin valoare

- Aceasta inseamna ca, atunci când se apelează o functie, ceea ce se transmite sunt valorile pe care le au argumentele în momentul apelului, valori care sunt **copiate** în variabilele reprezentate de parametrii funcției.
- Indiferent ce schimbari se fac asupra parametrilor functiei acestea nu influenteaza argumentele pe care le transmitem

- **Parametrii formali vs. acuali**
  - **Parametrii formali** – parametrii dati in definitia functiei
  - **Parametrii actuali** – parametrii folositi cand apelam functia (argumentele)

- Parametrii formali nu sunt variabile. O variabilă este caracterizată de nume, tip, şi adresă. Legarea unui parametru formal la o **adresă** se realizează în timpul execuţiei instrucţiunii de apelare a functiei.

- Deci, la **apelul prin valoare** se transmite o copie a parametrului actual. Valorile transmise la apelul unei functii sunt memorate în stivă. Datorită faptului că, după terminarea execuţiei functiei, stiva este eliberată, în cazul apelului prin valoare parametrul actual **nu se modifică.**
(se operează asupra unei copii a parametrului efectiv)

Apelul prin valoare

- Exemplu

```
void test_param (int formal) {      //formal este o copie a lui actual
        cout<<formal <<endl;        //50
        formal=100;                 //se va schimba doar copia
        cout<<formal<<endl;         //100
}

int main( ){
        int actual {50};
        cout<<actual<<endl;                         //50
        test_param (actual);        // se transmite o copie a lui actual ca argument functiei test_param
        cout<<actual<<endl;
        return 0;                    //50 (actual nu isi schimba valoarea)
}
```

Instructiunea return

- Daca o functie returneaza o valoare atunci **trebuie** sa folosim instructiunea **return** care sa returneze acea valoare
- Daca o functie nu returneaza o valoare (*void*) atunci instructiunea **return** este **optionala**
- Instructiunea **return** poate fi scrisa oriunde in corpul functiei
- Instructiunea **return** termina imediat functia
- Putem avea mai multe instructiuni **return** intr-o functie
- Valoarea returnata este rezultatul apelului functiei
- Vezi programul “ApelPrinValoare”

Valorile implicite ale argumentelor

- Cand apelam o functie, toate argumentele trebuie furnizate
- Uneori anumite argumente **au aceeasi valoarea** in majoritatea timpului.
- Putem sa ii spunem compilatorului sa foloseasca **valorile implicite** daca argumentele nu sunt furnizate
- Valorile implicite/ **argumentele implicite** pot fi date in **prototipul functiilor** sau in **definitia functiilor**, nu in ambele
  - Cea mai buna practica este sa dam valorile implicite in prototipul functiilor
  - Apar la **sfarsitul listei parametrilor**
- Putem avea **multiple** valori implicite
  - Prin urmare, toate valorile implicite trebuie sa apara la sfarsitul listei parametrilor

Valorile implicite ale argumentelor

- Exemplu – fara valori implicite ale argumentelor

```
//Vrem sa calculam pretul cu TVA pentru un anumit produs, al carui pret fara TVA se cunoaste si stiind ca
//TVA reprezinta 19% din valoarea produsului

#include<iostream>

using namespace std;

double calc_pret_cu_TVA(double pret_fara_TVA, double taxa);

double calc_pret_cu_TVA(double pret_fara_TVA, double taxa) {
        return pret_fara_TVA += (taxa*pret_fara_TVA);
}

int main() {
        double pret{};
        pret = calc_pret_cu_TVA(100.0, 0.21);
        cout << "Pretul cu TVA este " << pret << endl;

        return 0;
}
```

121

Valorile implicite ale argumentelor

- Exemplu – cu o valoare implicita a unui argument

```
//Vrem sa calculam pretul cu TVA pentru un anumit produs, al carui pret fara TVA se cunoaste si stiind ca
//TVA reprezinta 19% din valoarea produsului

#include<iostream>

using namespace std;

double calc_pret_cu_TVA(double pret_fara_TVA, double taxa=0.21);

double calc_pret_cu_TVA(double pret_fara_TVA, double taxa) {
        return pret_fara_TVA += (taxa*pret_fara_TVA);
}

int main() {
        double pret{};
        pret = calc_pret_cu_TVA(100.0);
        cout << "Pretul cu TVA este " << pret << endl;

        return 0;
}
```

121

Valorile implicite ale argumentelor

- Exemplu – cu o valoare implicita a unui argument

```
#include<iostream>

using namespace std;

double calc_pret_cu_TVA(double pret_fara_TVA, double taxa=0.21);

double calc_pret_cu_TVA(double pret_fara_TVA, double taxa) {
        return pret_fara_TVA += (taxa*pret_fara_TVA);
}

int main() {
     double pret{};
     pret = calc_pret_cu_TVA(100.0);
     cout << "In Romania, pretul cu TVA este " << pret << endl;      121
     pret = calc_pret_cu_TVA(100.0, 0.20);
     cout << "In Moldova, pretul cu TVA este " << pret << endl;      120


    return 0;
}
```

Valorile implicite ale argumentelor

- Exemplu – cu doua valori implicite ale argumentelor

```
double calc_pret_cu_TVA_si_transport(double pret_fara_TVA,  double transport=7.5, double taxa = 0.19);

double calc_pret_cu_TVA_si_transport(double pret_fara_TVA,  double transport, double taxa) {
        return pret_fara_TVA += (taxa*pret_fara_TVA)+transport;
}

int main() {
        double pret{};
        pret = calc_pret_cu_TVA_si_transport(100.0);     //valoarea transportului si a taxei sunt cele implicite
        cout << "In Romania, pretul cu TVA si transport in tara este " << pret << endl;

        pret = calc_pret_cu_TVA_si_transport(100.0, 25.0);   //valoarea taxei este implicita
        cout << "In Romania, pretul cu TVA si transport in strainatate este " << pret << endl;

        pret = calc_pret_cu_TVA_si_transport(50.0, 5.5, 0.15); //nici o valoare implicita
        cout << "In Moldova, pretul cu TVA si transport este " << pret << endl;

        return 0;
}
```

126.5

144

63

Valorile implicite ale argumentelor

- Exemplu – cu toate valorile implicite ale argumentelor

```
double calc_pret_cu_TVA_si_transport(double pret_fara_TVA=100.0,  double transport=7.5, double taxa = 0.19);

double calc_pret_cu_TVA_si_transport(double pret_fara_TVA,  double transport, double taxa) {
        return pret_fara_TVA += (taxa*pret_fara_TVA)+transport;
}

int main() {
        double pret{};
        pret = calc_pret_cu_TVA_si_transport();     //toate valorile sunt cele implicite
        cout << "In Romania, pretul redus cu TVA si transport in tara este " << pret << endl;   126.5

        pret = calc_pret_cu_TVA_si_transport(200.0);   //valorile transportului si ale taxei sunt implicite
        cout << "In Romania, noul pret cu TVA si transport in strainatate este " << pret << endl;
                                                                                                245.5
        return 0;
}
```

Valorile implicite ale argumentelor

- Exemplu – EROARE!!!! (Valorile implicite trebuie puse la coada listei parametrilor)

```
double calc_pret_cu_TVA_si_transport(double pret_fara_TVA=100.0,  double transport, double taxa = 0.19);

double calc_pret_cu_TVA_si_transport(double pret_fara_TVA,  double transport, double taxa) {
        return pret_fara_TVA += (taxa*pret_fara_TVA)+transport;
}
```

Valorile implicite ale argumentelor

- Exemplu

```
#include<iostream>
#include<string>

using namespace std;

void salutare(string nume, string prefix = "Dl.", string sufix = ”!");

void salutare(string nume, string prefix, string sufix) {
        cout << "Buna ziua " << prefix + " " + nume + " " + sufix << endl;
}

int main() {
    salutare("Simona Barna", "Dr.", "Ph.D");
    salutare("Adrian Popescu", "Profesor");
    salutare("Stefan Cristea");
    return 0;
}
```

Buna ziua Dr. Simona Barna Ph.D

Buna ziua Profesor Adrian Popescu !

Buna ziua Dl. Stefan Cristea !

Supraincarcarea functiilor

- Putem avea functii care sa aiba parametrii diferiti dar sa aiba acelasi nume

- **Polimorfismul** este capacitatea unor entitati de a lua forme diferite (este unul din conceptele esentiale din POO)
  - **Polimorfism parametric**
  - **Polimorfism de mostenire**

- Compilatorul este capabil sa depisteze functia pe care dorim sa o folosim in functie de argumentele furnizate si sa ne determine rezultatul dorit.
- Este un mecanism de **abstractizare.**

Supraincarcarea functiilor

Exemplu

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

int calc_suma(int, int);
double calc_suma(double, double);

int main() {

cout << calc_suma(10, 20)<<endl;

cout << calc_suma(10.0, 20.0) << endl;

cout << calc_suma(10, 20.0) << endl;

return 0;
}
int calc_suma(int a, int b) {
return a + b;
}

double calc_suma(double a, double b) {
return a + b;
}
```

//30 (functia int calc_suma)

//30.0 (functia double calc_suma)

EROARE! Compilatorul nu este capabil sa aleaga una din cele doua functii

Supraincarcarea functiilor

- Atentie atunci cand folosim supraincarcarea functiilor si parametrii impliciti!

```cpp
#include<iostream>
#include<string>
#include<vector>
using namespace std;


//Atentie atunci cand folosim supraincararea functiilor si parametrii impliciti


void afisare(int n=10);
void afisare(double d=123.2);

void afisare(int n) {
    cout << "Afisare nr intreg:  " << n << endl;
}
void afisare(double d) {
    cout << "Afisare nr real:  " << d << endl;
}

int main() {

    afisare();
    return 0;
}
```

EROARE!!!!

Supraincarcarea functiilor

- Exista o restrictie in ceea ce priveste supraincarcarea functiilor. **Tipul returnat nu este considerat**

Exemplul 1.

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

double suma(int a, int b) {
return (a + b) / 2.0;
}

cout<<suma(3,5)<<endl;                    //Eroare
```

Parametrii functiilor - tablouri

- Putem avea drept parametrii ai unei functii tablouri unidimensionale /multidimensionale

Exemplu:

```
void afiseaza_tablou (int numere[ ]);
```

- Elementele tabloului **nu** sunt copiate!

- Din moment ce numele tabloului reprezinta adresa din memorie a tabloului (a primului element din tablou), adresa este cea care este copiata
- Prin urmare, functia nu are nici o idee despre cate elemente sunt in tablou din moment ce tot ceea ce stie este locatia primului element (numele tabloului)

Parametrii functiilor - tablouri

- Exemplu:

```
void afiseaza_tablou(int numere[]);

int main() {
	int numerele_mele[]{ 1,2,3,4,5 };
	afiseaza_tablou(numerele_mele);
	return 0;
}

void afiseaza_tablou(int numere[]) {
	// compilatorul nu stie cate elemente are tabloul
	// trebuie sa ii transmitem compilatorului aceasta informatie,
	// deci va trebui sa oferim ca argument si dimensiunea tabloului

}
```

Parametrii functiilor - tablouri

- Exemplu:

```
void afiseaza_tablou(int numere[], size_t dim);

int main() {
int numerele_mele[]{ 1,2,3,4,5 };
afiseaza_tablou(numerele_mele,5);
return 0;
}

void afiseaza_tablou(int numere[], size_t dim) {
for (size_t i{ 0 }; i < dim;i++)
cout << numere[i] << endl;
}
```

```
1
2
3
4
5
```

Parametrii functiilor - tablouri

- Trebuie sa fim atenti la faptul ca din moment ce atunci cand dam ca parametru un tablou, ceea ce se copiaza este de fapt, adresa tabloului, noi **putem modifica** in cadrul functiei valorile elementelor tabloului, asa cum se poate observa si in exemplul urmator:

Parametrii functiilor - tablouri

```
void afiseaza_tablou(int numere[], size_t dim);
void tablou_nul(int numere[], size_t dim);

int main() {
        int numerele_mele[]{ 1,2,3,4,5 };
        cout << "Tabloul initial este : " << endl;
        afiseaza_tablou(numerele_mele,5);
        tablou_nul(numerele_mele, 5);
        cout << "\nTabloul dupa apelul functiei tablou_nul este:
        " << endl;
        afiseaza_tablou(numerele_mele, 5);

        return 0;
}

void afiseaza_tablou(int numere[], size_t dim) {
        for (size_t i{ 0 }; i < dim;i++)
        cout << numere[i] << endl;
}
void tablou_nul(int numere[], size_t dim) {
        for (size_t i{ 0 }; i < dim;i++)
        numere[i] = 0;
}
```

Tabloul initial este:
1
2
3
4
5

Tabloul dupa apelul functiei tablou_nul este:

0
0
0
0
0

Parametrii functiilor - tablouri

- Daca nu dorim sa schimbam valorile elementelor din tablou avem urmatoarea posibilitate
- Putem spune compilatorului ca parametrii functiilor sunt constanti

```
void afiseaza_tablou(const int numere[], size_t dim) {
    for (size_t i{ 0 }; i < dim;i++)
    {
        cout << numere[i] << endl;
        numere[i] = 0; // orice incercare de a modifica tabloul va
                       // genera o eroare a compilatorului
    }

}
```

- Vezi programul “Parametrii-tablouri”!

Parametrii functiilor - tablouri

- **Aplicaţii -- Căutarea secvenţială**

Căutarea secvenţială este unul dintre cei mai simpli algoritmi studiaţi. El urmăreşte să verifice apartenenţa unui element la un şir de elemente de aceeaşi natură, în speţă a unui număr la un şir de numere. Pentru aceasta:

- Se parcurge şirul de la un capăt la celălalt şi se compară numărul căutat cu fiecare număr din şir.
- In cazul în care s-a găsit corespondenţă (egalitate), un indicator flag este poziţionat.
- La sfârşitul parcurgerii şirului, indicatorul ne va arăta dacă numărul căutat aparţine sau nu şirului.

Parametrii functiilor - tablouri

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

void CautareSecventiala(int tab[], int n, int x) {
    bool gasit{ 0 };
    for (int i{ 0 }; i < n; i++) {
        if (tab[i] == x) gasit = 1;
    }
    if (gasit == 0)
        cout << "Numarul nu apartine sirului !" << endl;
    else cout << "Numarul apartine sirului !" << endl;
}
int main()
{
    int n, x, tab[50];
    cout << "Dati numarul de elemente ale sirului : "; cin >> n;
    cout << "Dati numarul de cautat : "; cin >> x;
    cout << "Dati elementele sirului" << endl;
    for (int i{ 0 }; i < n; i++) {
        cout << "tab[" << i << "]= ";
        cin >> tab[i];
    }
    CautareSecventiala(tab, n, x);
    return 0;
}
```

Parametrii functiilor - tablouri

Aplicaţii -- Căutare binară

Algoritmul de **căutare binară** oferă performanţe mai bune decât algoritmul de căutare secvenţială. El funcţionează astfel:

- Se compară numărul căutat cu elementul aflat la mijlocul şirului (element care se mai numeşte şi **pivot**).
- În cazul în care cele două elemente coincid căutarea s-a încheiat cu succes. Dacă numărul căutat este mai mare decât pivotul, se continuă căutarea în aceeaşi manieră în subşirul delimitat de pivot şi capătul şirului iniţial. Dacă numărul căutat este mai mic decât pivotul se continuă căutarea în aceeaşi manieră în subşirul delimitat de pivot şi începutul şirului iniţial.

Algoritmul prezentat se încadrează în clasa algoritmilor elaboraţi conform tehnicii de programare *Divide et Impera*. Unul din **dezavantajele acestui algoritm** este că şirul în care se face căutarea trebuie să fie initial **sortat**.

Parametrii functiilor - tablouri

```cpp
#include<iostream>
using namespace std;
void cautareBinara(int tab[], int n, int x) {
    int st, dr, mijl, gasit;
    st = 0; dr = n - 1; gasit = 0;
    while (st <= dr && gasit != 1) {
        mijl = (st + dr) / 2;
        if (tab[mijl] == x)
            gasit = 1;
        else
            if (x < tab[mijl])
                dr = mijl - 1;
            else
                st = mijl + 1;
    }
    if (st > dr)
        cout << "Nu s-a gasit elementul cautat !" << endl;
    else
        cout << "Elementul cautat apartine sirului !" << endl;

}
int main()
{
    int n,x, tab[50];
    cout << "Introduceti numarul elementelor vectorului : ";
    cin >> n;
    cout << "Introduceti elementul de cautat : ";
    cin >> x;
    cout << "Introduceti elementele vectorului (in ordine crescatoare):" << endl;
    for (int i = 0; i < n; i++) {
        cout << "tab[" << i << "]= ";
        cin >> tab[i];
    }
    cautareBinara(tab, n, x);
    return 0;
}
```

Debug:1,3,5,7,9,10,13; x=13

Parametrii functiilor - tablouri

Aplicaţii -- Sortarea prin metoda bulelor

- Acest algoritm se mai numeşte şi "**sortarea prin selecţie şi interschimbare**", "**sortarea prin propagare**" sau "**metoda lentă de sortare**" datorită numărului mare de operaţii care trebuie efectuate. Succesul algoritmului este asigurat de trecerea succesivă prin tablou, până când acesta este sortat, cu specificaţia că, la fiecare trecere, elementele succesive i şi i+1 pentru care **tab[i]>tab[i+1]**, vor fi interschimbate.
- Metoda poate fi imbunătăţită dacă, după fiecare trecere, se va reţine ultima poziţie din tablou în care a avut loc o interschimbare, iar trecerea următoare se va efectua doar pană la acea poziţie. În cazul în care la o trecere nu a avut loc nici o interschimbare algoritmul se va incheia.
- Pentru o şi mai bună optimizare se poate înlocui trecerea prin tablou într-un sens cu trecerea în dublu sens. În acest caz, dacă la două treceri succesive, între două elemente i şi i+1 nu a avut loc o interschimbare, atunci nici la trecerile următoare nu se vor înregistra interschimbări.
- Cu toate optimizările aduse acestei sortari, ea se dovedeşte a fi cu **mult mai lentă** decât **sortarea prin inserţie**, fiind însă preferată de unii datorită simplităţii, ce nu ridică probleme de implementare. Pentru valori suficient de mari ale dimensiunii vectorului ce urmează a fi sortat se recomandă utilizarea unor algoritmi ce au o complexitate mai redusa şi, prin urmare, oferă un timp de calcul mai mic.

Parametrii functiilor - tablouri

Aplicaţii -- Sortarea prin metoda bulelor

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

void SortareMetodaBulelor(int tab[], int n) {
    int aux, terminat = 0;
    while (terminat==0) {
        terminat = 1;
        for (int i = 0; i < n - 1; i++)
            if (tab[i] > tab[i + 1]) {
                aux = tab[i];
                tab[i] = tab[i + 1];
                tab[i + 1] = aux;
                terminat = 0;
            }
    }
    cout << "Vectorul ordonat este : ";
    for (int i = 0; i <= n - 1; i++)
        cout << tab[i] << " ";
    cout << endl;
}

int main(){

int n, i, tab[50];
cout << "Introduceti dimensiunea vectorului : ";
cin >> n;
for (i = 0; i <= n - 1; i++) {
    cout << "tab[" << i << "]=";
    cin >> tab[i];
}
SortareMetodaBulelor(tab, n);
return 0;
}
```

Debug pentru tab={3,8,5,4,1}

Parametrii functiilor - tablouri

Aplicaţii -- Sortarea prin inserţie

- se bazează pe aceleaşi principii ca şi cele aplicate de majoritatea jucătorilor de cărţi, adică după ridicarea unei cărţi de pe masă, aceasta se aşează în pachetul din mână la locul potrivit. Cu alte cuvinte, considerăm că avem **vectorul sortat** tab, iar la ivirea unui nou element care se va adăuga vectorului, el va fi pus pe locul potrivit printr-o inserţie în interiorul vectorului.
- **Inserţia/inserarea directă**. Este cea mai simplă implementare a algoritmului şi se face în felul următor: Se consideră că primele i elemente al vectorului sunt deja sortate. Pentru elementul al (i+1)-lea, din tabloul iniţial, se **va cauta secvential** poziţia în care trebuie inserat printre primele i elemente. Toate elementele tabloului de la această poziţie şi până la i vor fi deplasate cu o poziţie mai la dreapta iar poziţia eliberată va fi ocupată de elementul i+1.
- **Inserţia /inserarea rapidă**. Deoarece vectorul destinaţie este un vector ordonat crescător, căutarea poziţiei în care va fi inserat tab[i] se poate face nu secvenţial (ca în cazul inserării directe) ci prin algoritmul de **căutare binară**. Subvectorul destinaţie este împărţit în doi subvectori, se examinează relaţia de ordine dintre elementul de la mijlocul subvectorului şi elementul tab[j] şi se stabileşte dacă elementul tab[i] va fi inserat în prima jumătate sau în a doua jumătate. Operaţia de divizare a subvectorului continuă până se găseşte poziţia în care urmează să fie inserat tab[i].

Parametrii functiilor - tablouri

Aplicaţii -- Sortarea prin inserţie directa

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

void SortarePrinInsertie(int tab[], int n) {
    int aux,i;
    for (int j = 1; j < n; j++) {
        aux = tab[j];
        i = j - 1;
        while (aux < tab[i] && i >= 0) {
            tab[i + 1] = tab[i];
            i = i - 1;
        }
        tab[i + 1] = aux;
    }
    cout << "Sirul ordonat este: ";
    for (i = 0; i < n; i++)
        cout << tab[i] << " ";
    cout << endl;
}

int main() {

    int n, i, tab[50]{};
    cout << "Introduceti dimensiunea vectorului : ";
    cin >> n;
    for (i = 0; i <= n - 1; i++) {
        cout << "tab[" << i << "]=";
        cin >> tab[i];
    }
    SortarePrinInsertie(tab, n);
    return 0;
}
```

Debug pentru tab={3,8,5,4,1}

Parametrii functiilor - tablouri

Aplicaţii -- Sortarea prin inserţie rapida

```
void QuickSort(int v[], int st, int dr)
{
    if (st < dr)
    {
        //pivotul este inițial v[st]
        int m = (st + dr) / 2;
        int aux = v[st];
        v[st] = v[m];
        v[m] = aux;
        int i = st, j = dr, d = 0;
        while (i < j)
        {
            if (v[i] > v[j])
            {
                aux = v[i];
                v[i] = v[j];
                v[j] = aux;
                d = 1 - d;
            }
            i =i+ d;
            j = j-(1 - d);
        }
        QuickSort(v, st, i - 1);
        QuickSort(v, i + 1, dr);
    }
}
```

```
int main() {

    int n, i, tab[50]{};
    cout << "Introduceti dimensiunea vectorului : ";
    cin >> n;
    for (i = 0; i <= n - 1; i++) {
        cout << "tab[" << i << "]=";
        cin >> tab[i];
    }
    QuickSort(tab, 0,n-1);

    cout << "Sirul ordonat este: ";
    for (int i = 0; i < n; i++)
        cout << tab[i] << " ";
    cout << endl;
    return 0;
}
```

Debug pentru tab={3,8,5,4,1}

Parametrii functiilor - tablouri

Aplicaţii -- Sortarea prin numărare

Această metodă necesită spaţiu suplimentar de memorie. Ea foloseşte următorii vectori:
- vectorul sursă (vectorul nesortat) a;
- vectorul destinaţie (vectorul sortat) b;
- vectorul numărător (vectorul de contoare) k.

Elementele vectorului sursă a[i] se copie în vectorul destinaţie prin inserarea în poziţia corespunzătoare, astfel încât, în vectorul destinaţie să fie respectată relaţia de ordine. Pentru a se cunoaşte poziţia în care se va insera fiecare element, se parcurge vectorul sursă şi se numără în vectorul k (**vector de frecventa**), pentru fiecare element a[i], câte elemente au valoarea mai mică decât a lui. Fiecare element al vectorului k este un contor pentru elementul a[i]. Valoarea contorului k[i] pentru elementul a[i] reprezintă câte elemente sunt mai mici decat el şi arată de fapt poziţia în care trebuie copiat în vectorul b.

Parametrii functiilor - tablouri

Aplicaţii -- Sortarea prin numărare

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

void SortarePrinNumarare(int a[], int n) {
    int i, j, b[50], k[50] {};
    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j])
                k[i]++;
            else k[j]++;
    for (i = 0; i < n; i++)
        b[k[i]] = a[i];
    cout << "Vectorul ordonat este : ";
    for (i = 0; i < n; i++)
        cout << b[i] << " ";
    cout << endl;

}

int main() {

    int n, i, tab[50];
    cout << "Introduceti dimensiunea vectorului : ";
    cin >> n;
    for (i = 0; i <= n - 1; i++) {
        cout << "tab[" << i << "]=";
        cin >> tab[i];
    }
    SortarePrinNumarare(tab, n);
    return 0;
}
```

Debug pentru tab={3,8,5,4,1}

Apelul prin referinta

- Am vazut ca atunci cand apelam o functie, in mod implicit apelarea se face prin valoare, adica se face o copie a parametrului
- De asemenea, am vazut ca atunci cand parametrii sunt niste tablouri nu se mai creaza copii ale parametrilor, in schimb se foloseste locatia tablourilor si prin urmare putem schimba valorile elementelor tabloului in cadrul functiei

- Uneori dorim sa putem fi capabili sa schimbam parametrii actuali (argumentele) in corpul functiei
- Pentru a putea face acest lucru avem nevoie de locatia (adresa) parametrului actual

- Am vazut cum se desfasoara lucrurile in cazul tablourilor, dar ce se intampla cand avem alte tipuri de variabile?

- Putem folosi **operatorul de referinta &** pentru a-i spune compilatorului sa foloseasca locatia parametrului actual
- Parametrul formal va fi acum un **alias** al parametrului actual

Apelul prin referinta

- Exemplu:

```
void test_numar(int &num);

void test_numar(int &num) {
    if (num > 100)
    num = 100;
    else
    num = 50;
}

int main() {
    int numar{ 1000 };
    cout << "Numarul initial este: " << numar << endl;                    1000
    test_numar(numar);
    cout << "Numarul dupa apelul functiei este: " << numar << endl;       100

return 0;
}
```

Apelul prin referinta

- Exemplu:

```
void schimba(int &a, int &b);

int main() {
    int x{ 10 }, y{ 20 };
    cout << x << " " << y << endl;
    schimba(x, y);
    cout << x << " " << y << endl;
    return 0;
}

void schimba(int &a, int &b) {
    int aux = a;
    a = b;
    b = aux;
}
```

10 20

20 10

Ce s-ar afisa daca nu folosim op. de referinta “&”?

Exemplu vector --- apel prin valoare

```
void afisare(vector <int> v);

int main() {
    vector<int> note{ 10,9,8 };
    afisare(note);
    return 0;
}

void afisare(vector <int> v) {
    for (auto num : v)
    cout << num << endl;
}
```

- De ce sa cream o copie a parametrului si sa folosim un spatiu din memorie pentru aceasta copie?

Exemplu vector --- apel prin referinta

```
void afisare(vector <int> &v);

int main() {
    vector<int> note{ 10,9,8 };
    afisare(note);
    return 0;
}

void afisare(vector <int> &v) {
    for (auto num : v)
    cout << num << endl;
}
```

- Nu ne convine totusi faptul ca in functia de afisare putem modifica valorile elementelor vectorului!

Exemplu vector --- apel prin referinta const

```
void afisare(const vector <int>  &v);

int main() {
    vector<int> note{ 10,9,8 };
    afisare(note);
    return 0;
}

void afisare( const vector <int> &v) {
    v[0] = 200;                    //EROARE
    for (auto num : v)
    cout << num << endl;
}
```

- Vezi programul “ApelPrinReferinta”!

Vizibilitatea entitatilor

- Vrem sa determinam cat timp si unde putem folosi identificatori declarati in program

- Partea din program in care o anumita entitate este valida poate fi:
  - Locala -> Locala statica
  - Globala

Vizibilitatea locala

- Identificatorii sunt declarati intr-un bloc { }
- Parametrii functiilor sunt vizibili doar acolo unde este declarata functia
- Variabilele locale declarate in cadrul unei functii sunt active doar cat timp se executa functia
- Variabilele locale nu se pastreaza intre apelurile functiilor
- Cand avem blocuri imbricate, cele din interior pot “vedea” entitatile declarate in cele din exterior, dar cele din exterior nu pot “vedea” in cele din interior

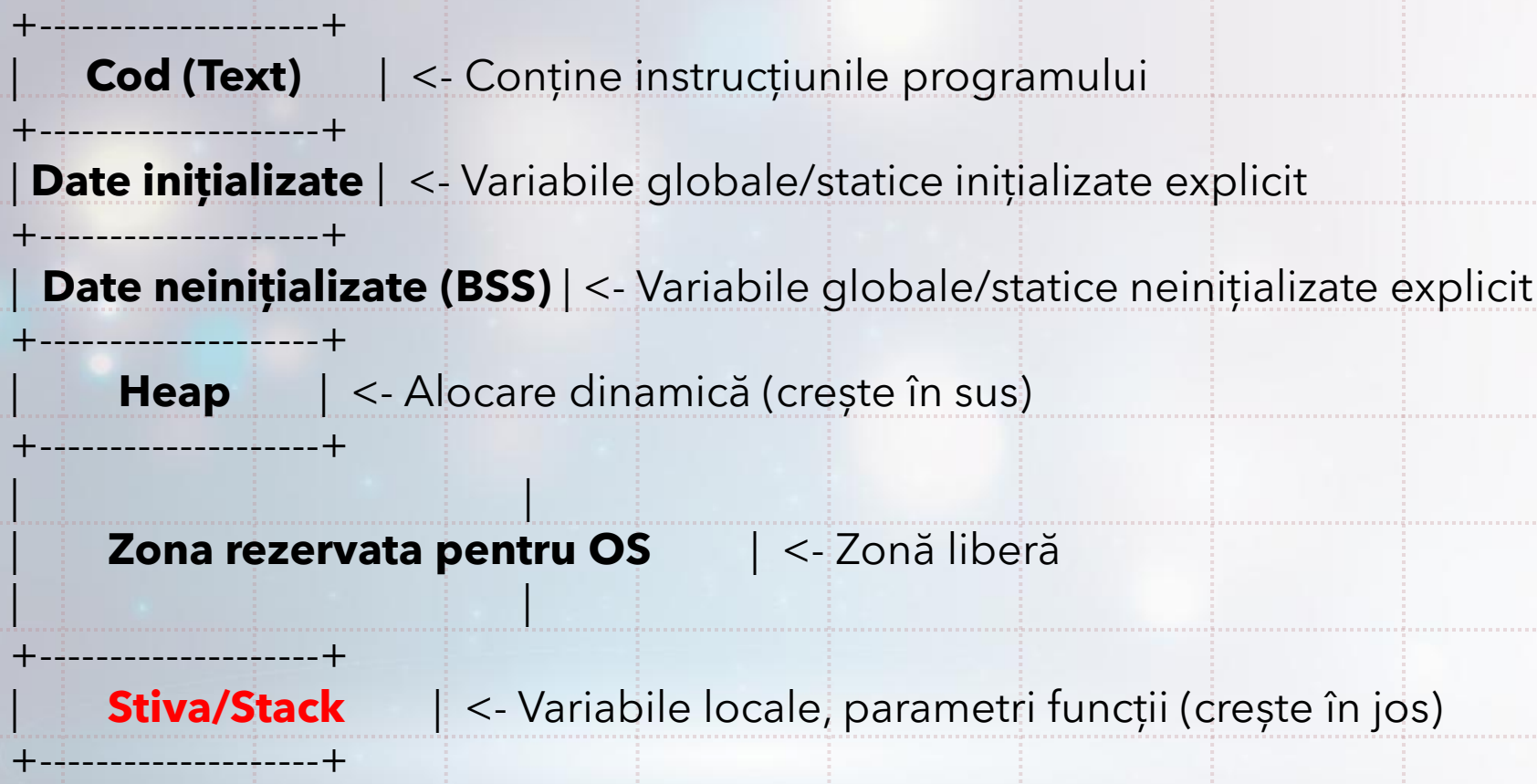
Variabile locale statice

- Se declara cu ajutorul cuvantului cheie **static**
  static int valoare {10};
- **Valoarea lor se pastreaza intre apelurile functiilor**
- “Viata” acestor variabile este pe durata intregului program, dar sunt vizibile doar in cadrul functiei in care sunt declarate
- Se initializeaza cand functia este apelata pentru prima data, iar daca nu se initializeaza explicit li se atribuie automat valoarea 0.
- Sunt foarte folositoare atunci cand vrem sa stim valorile lor precedente, fara a trebui sa transmitem de fiecare data aceste informatii.

Vizibilitatea globala

- Identificatorii sunt declarati in afara oricarei functii (sau clase)
- Dupa ce sunt declarati, identificatorii globali sunt vizibili in toate partile programului
- Constante globale – ok
- Recomandare: nu folositi variabile globale
- Vezi programul “Vizibilitatea”!

MEMORIA UNUI PROGRAM



Memoria unui program

- **Segmentul de cod (Text Segment)** contine codul executabil al programului, adică instrucțiunile mașinii generate de compilator.

- **Segmentul de date statice (Data Segment)** conține datele statice și variabilele globale care au fost inițializate înainte de rularea programului. Este împărțit în:
  - **Secțiunea de date inițializate:** pentru variabilele globale și statice care au valori inițiale explicite.
  - **Secțiunea de date neinițializate (BSS - Block Started by Symbol):** pentru variabile globale și statice care nu au fost inițializate explicit. Acestea sunt inițializate implicit la 0.

- **Segmentul de rezervare (Reserves/OS Reserved):** este o zonă rezervată de sistemul de operare pentru gestionarea internă a proceselor și comunicației.

Memoria unui program

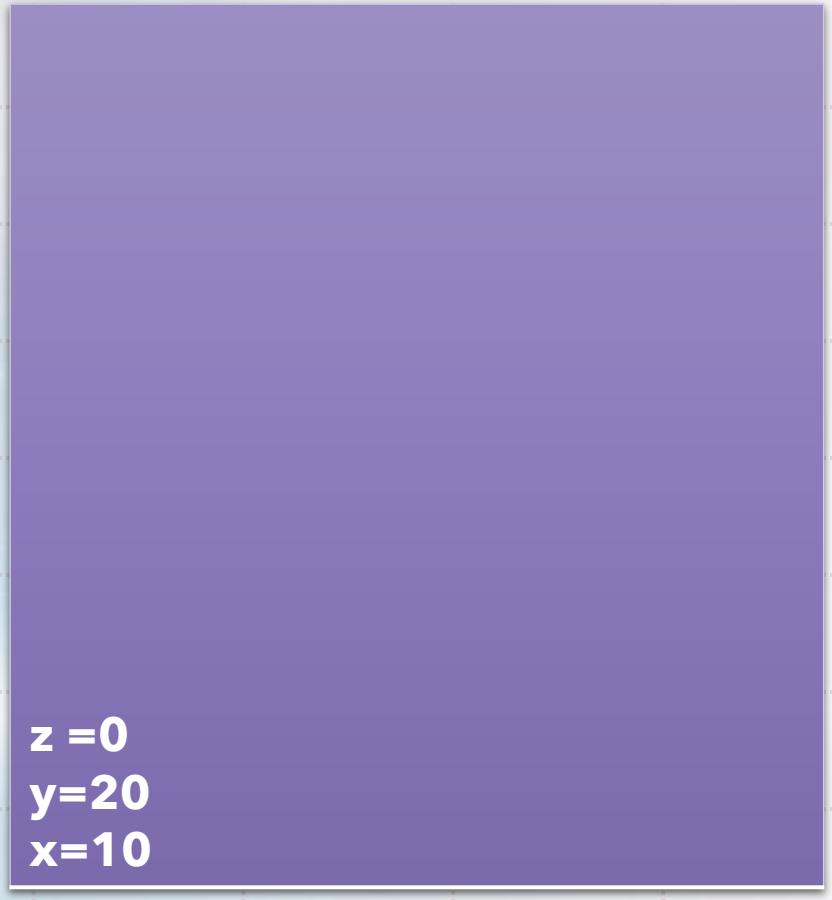
- **Stiva (stack)** este o zonă de memorie utilizată pentru gestionarea automată a variabilelor și a informațiilor asociate execuției unui program, cum ar fi **apelurile de funcții**. Este o structură organizată în mod **LIFO** (Last In, First Out), ceea ce înseamnă că ultima variabilă sau funcție introdusă este și prima care este eliminată.
- Analogul unei stive de carti. De fiecare data cand o functie este apelata se adauga in stiva o noua inregistrare, iar cand functia se termina, inregistrarea este scoasa din stiva.
- Dimensiunea stivei este finita, deci daca se apeleaza prea multe functii atunci se poate depasi acesta dimensiune (**Stack overflow**)

- Memoria **heap** este o zonă din memorie utilizată pentru de date în timpul execuției unui program. Spre deosebire de stiva (stack), unde alocarea și eliberarea memoriei sunt gestionate automat (de exemplu, pentru variabile locale și apeluri de funcții), memoria heap oferă control programatorului pentru alocarea și eliberarea manuală a spațiului necesar.
- Memoria heap este accesata prin **pointeri**. În general, heap-ul este mai mare decât stiva și poate gestiona volume mari de date.

Cum functioneaza apelul functiilor?

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

void functie2(int &x, int y, int z) {
        x += y + z;
}

int functie1(int a, int b) {
        int rezultat{};
        rezultat = a + b;
        functie2(rezultat, a, b);
        return rezultat;
}

int main() {
        int x{ 10 };
        int y{ 20 };
        int z{};
        z = functie1(x, y);
        cout << z << endl;
        return 0;
}
```

Stiva

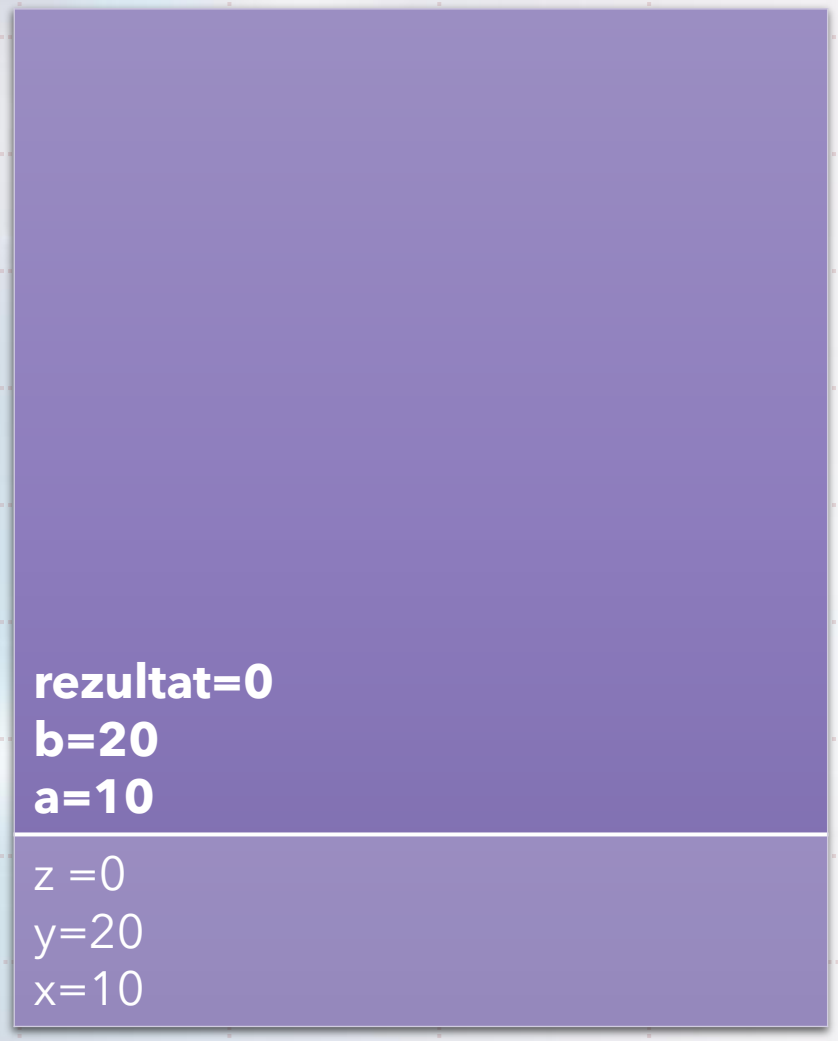


main

Cum functioneaza apelul functiilor?

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

void functie2(int &x, int y, int z) {
        x += y + z;
}

int functie1(int a, int b) {
        int rezultat{};
        rezultat = a + b;
        functie2(rezultat, a, b);
        return rezultat;
}

int main() {
        int x{ 10 };
        int y{ 20 };
        int z{};
        z = functie1(x, y);
        cout << z << endl;
        return 0;
}
```

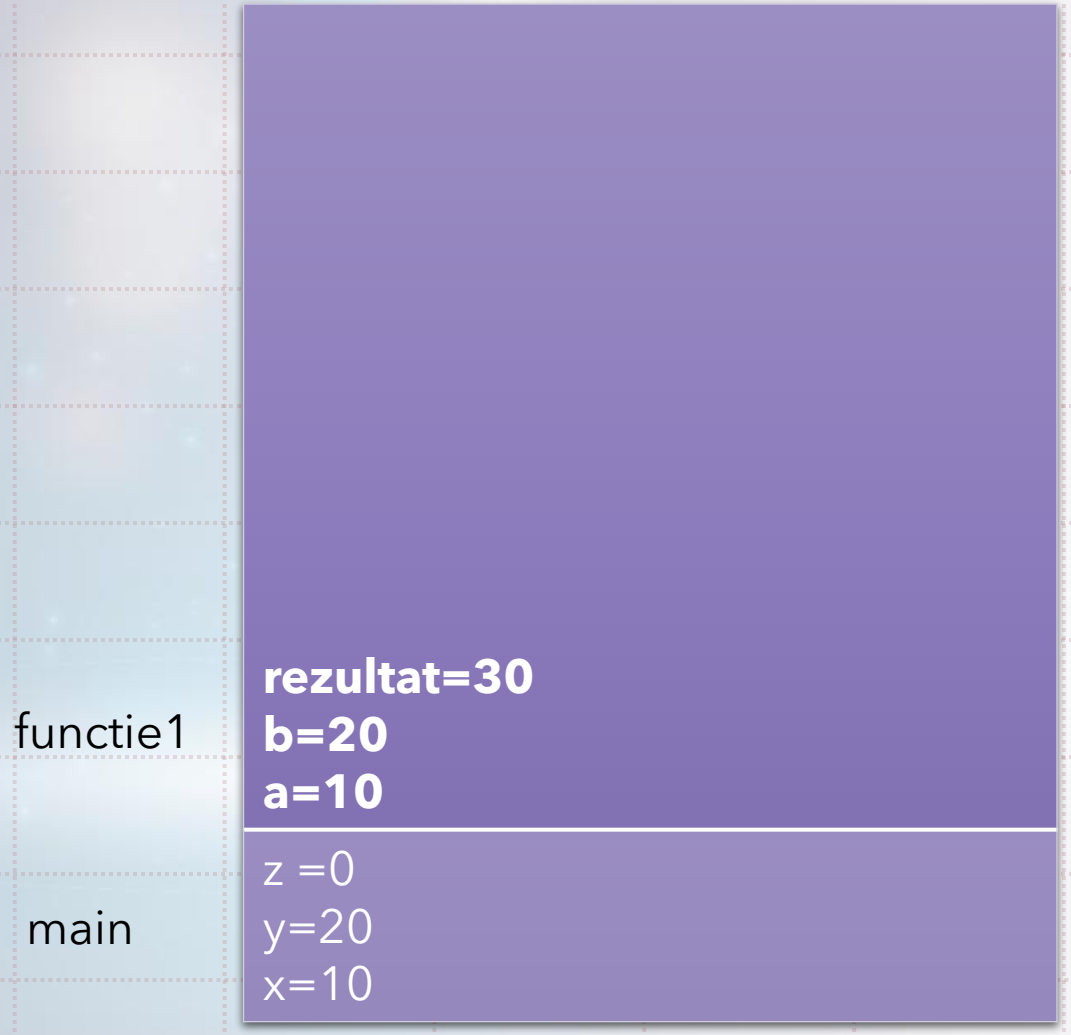
Stiva



Cum functioneaza apelul functiilor?

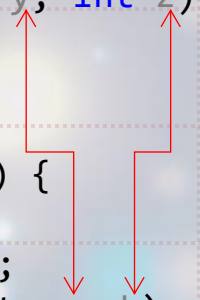
```
#include<iostream>
using namespace std;

void functie2(int &x, int y, int z) {
        x += y + z;
}

int functie1(int a, int b) {
        int rezultat{};
        rezultat = a + b;
        functie2(rezultat, a, b);
        return rezultat;
}

int main() {
        int x{ 10 };
        int y{ 20 };
        int z{};
        z = functie1(x, y);
        cout << z << endl;
        return 0;
}
```

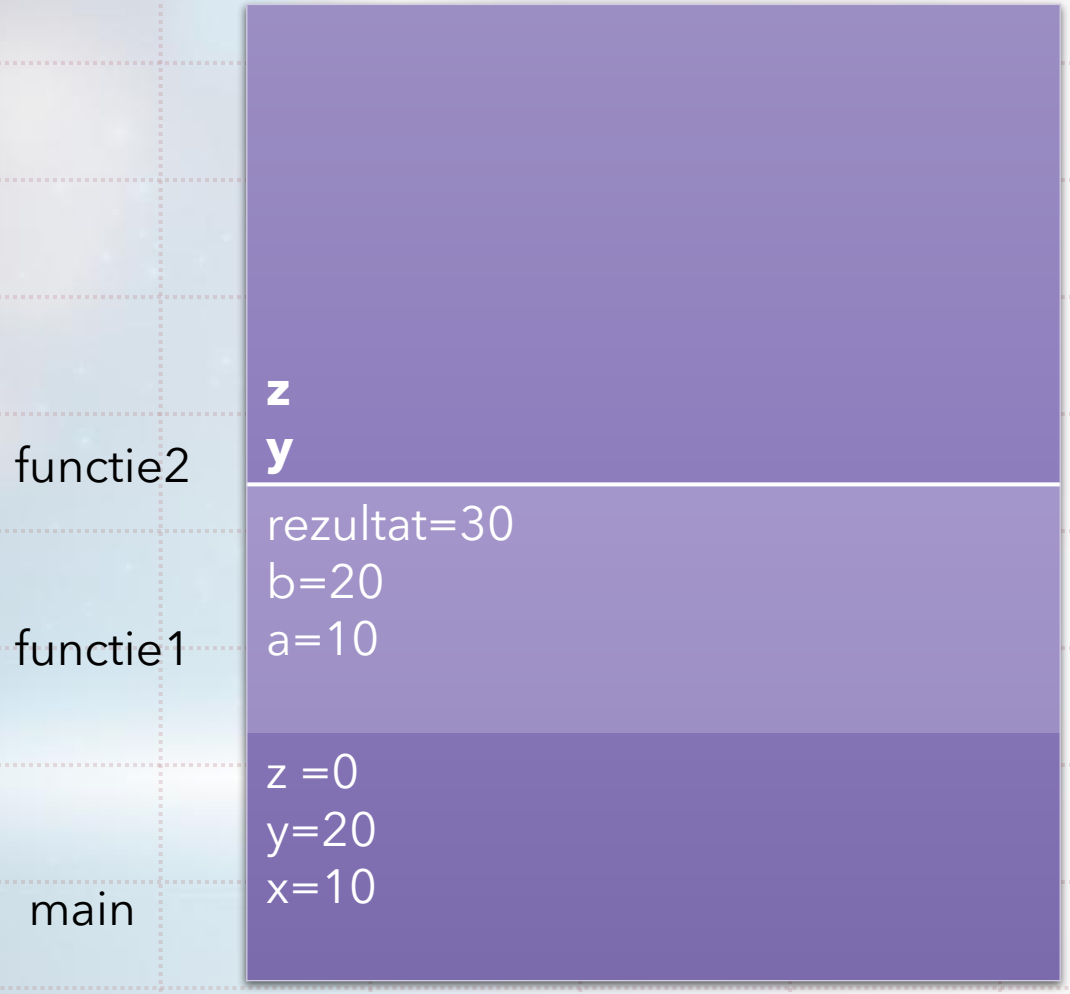
Stiva



Cum functioneaza apelul functiilor?

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

void functie2(int &x, int y, int z) {
        x += y + z;
}

int functie1(int a, int b) {
        int rezultat{};
        rezultat = a + b;
        functie2(rezultat, a, b);
        return rezultat;
}

int main() {
        int x{ 10 };
        int y{ 20 };
        int z{};
        z = functie1(x, y);
        cout << z << endl;
        return 0;
}
```

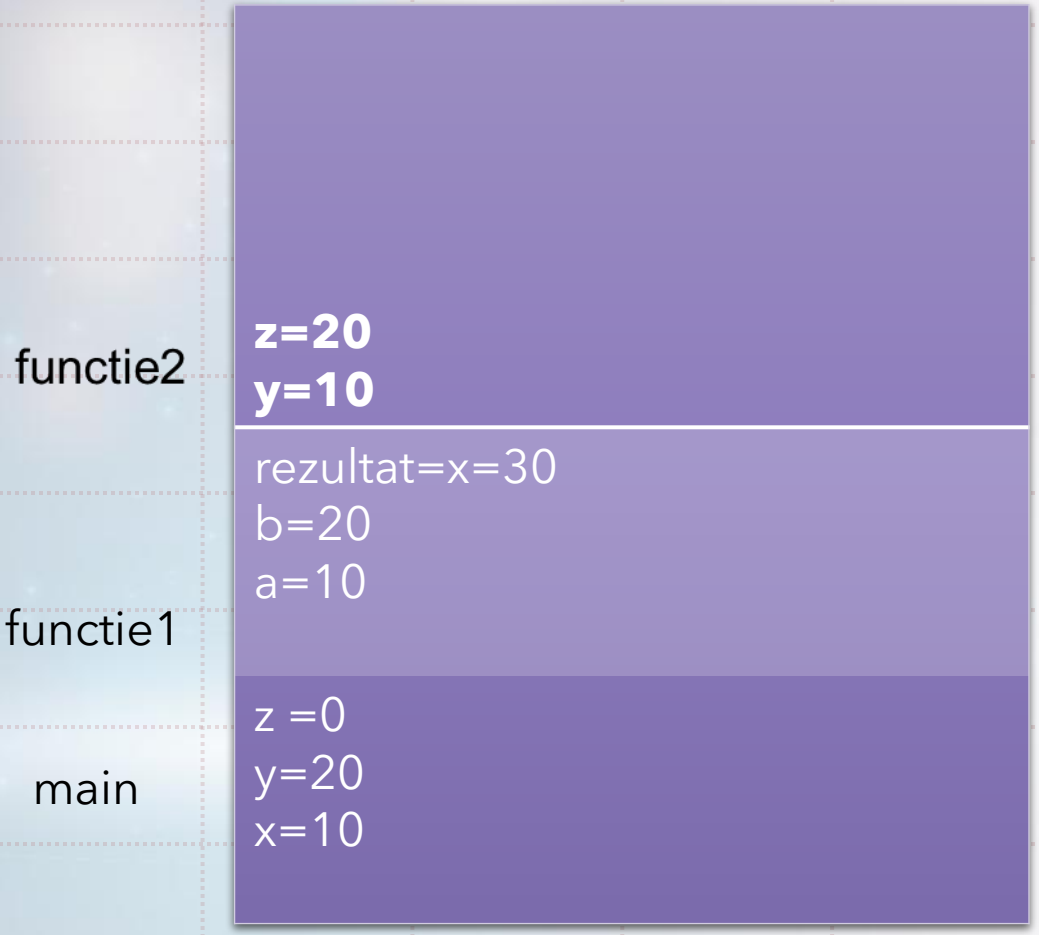




Cum functioneaza apelul functiilor?

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

void functie2(int &x, int y, int z) {
        x += y + z;
}

int functie1(int a, int b) {
        int rezultat{};
        rezultat = a + b;
        functie2(rezultat, a, b);
        return rezultat;
}

int main() {
        int x{ 10 };
        int y{ 20 };
        int z{};
        z = functie1(x, y);
        cout << z << endl;
        return 0;
}
```

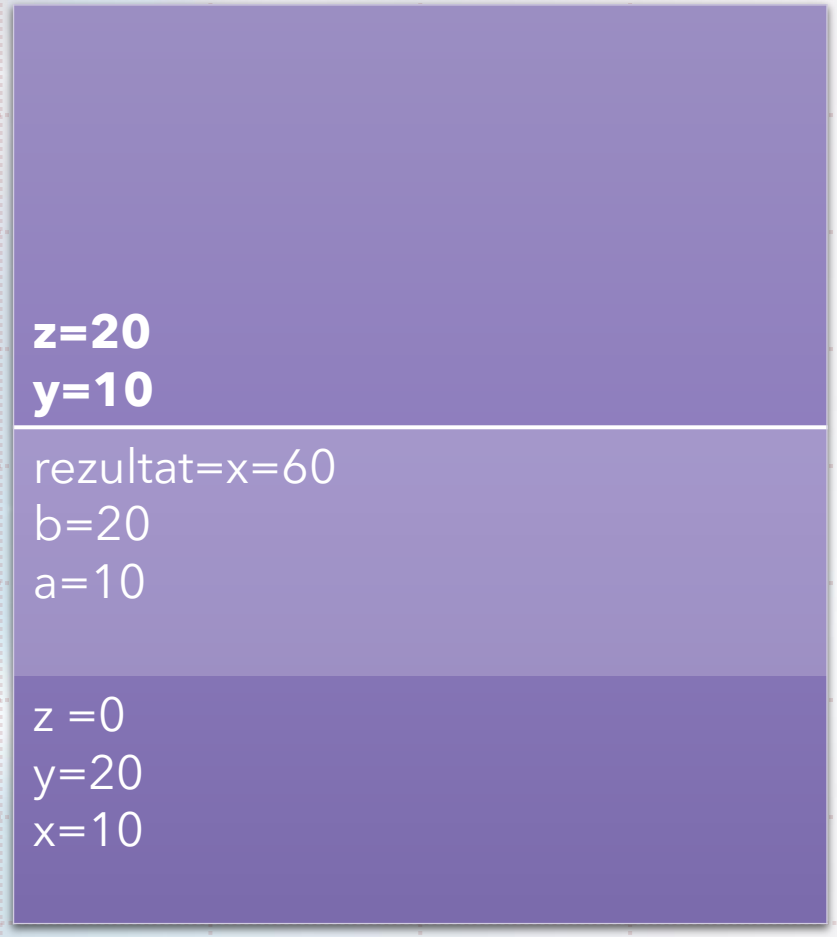
Stiva



Cum functioneaza apelul functiilor?

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

void functie2(int &x, int y, int z) {
        x += y + z;
}

int functie1(int a, int b) {
        int rezultat{};
        rezultat = a + b;
        functie2(rezultat, a, b);
        return rezultat;
}

int main() {
        int x{ 10 };
        int y{ 20 };
        int z{};
        z = functie1(x, y);
        cout << z << endl;
        return 0;
}
```

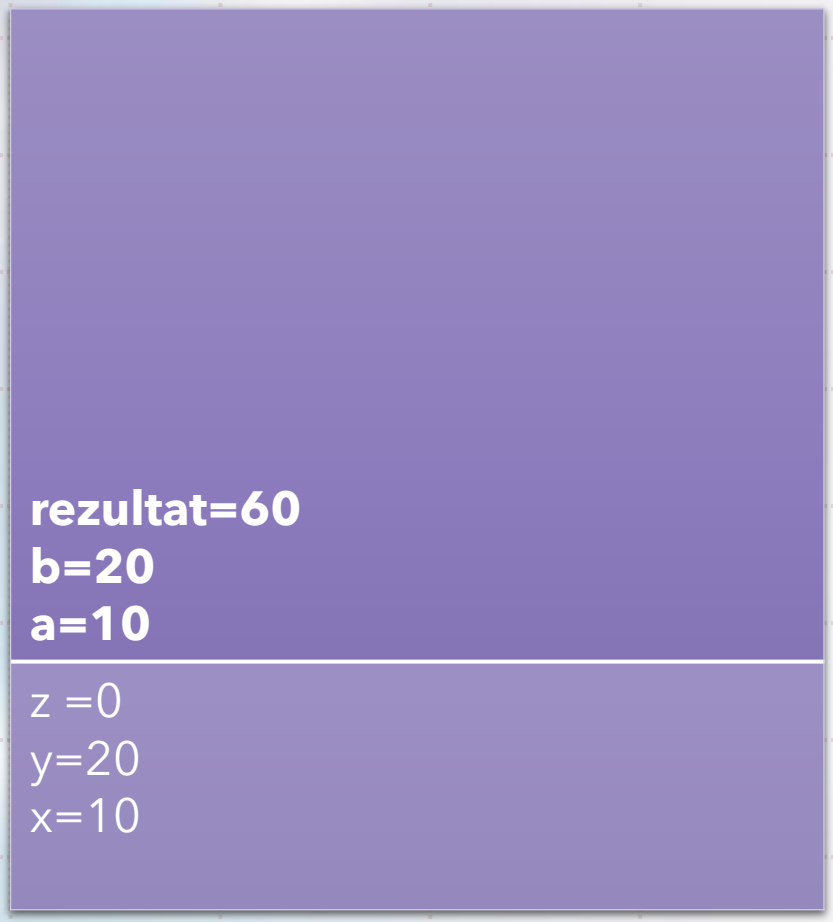
Stiva



Cum functioneaza apelul functiilor?

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

void functie2(int &x, int y, int z) {
        x += y + z;
}

int functie1(int a, int b) {
        int rezultat{};
        rezultat = a + b;
        functie2(rezultat, a, b);
        return rezultat;
}

int main() {
        int x{ 10 };
        int y{ 20 };
        int z{};
        z = functie1(x, y);
        cout << z << endl;
        return 0;
}
```

Stiva



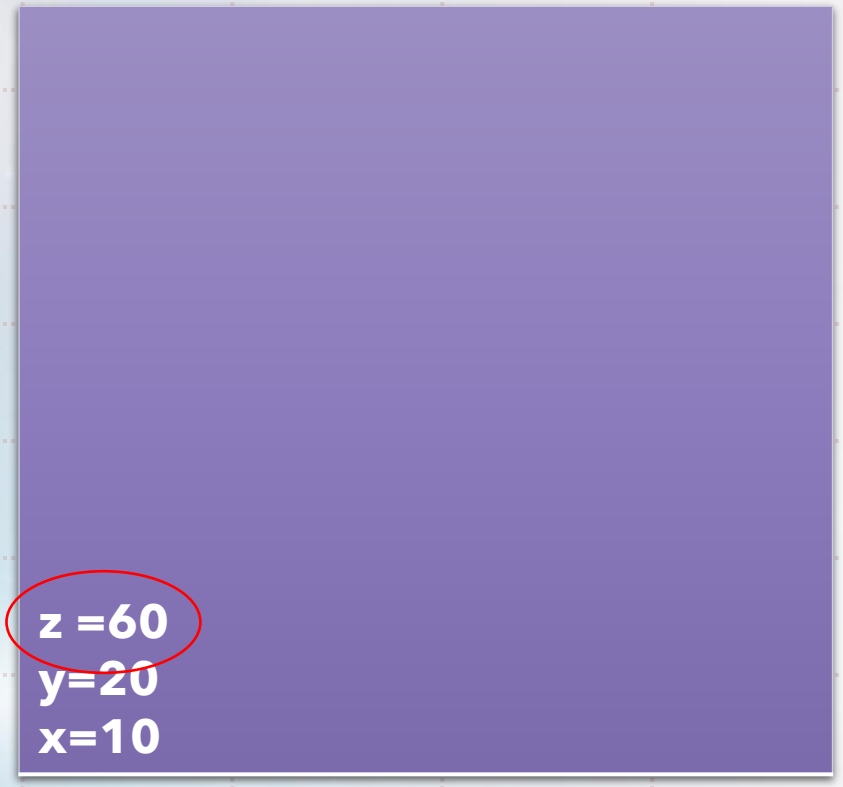
functie1

main

Cum functioneaza apelul functiilor?

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

void functie2(int &x, int y, int z) {
        x += y + z;
}

int functie1(int a, int b) {
        int rezultat{};
        rezultat = a + b;
        functie2(rezultat, a, b);
        return rezultat;
}

int main() {
        int x{ 10 };
        int y{ 20 };
        int z{};
        z = functie1(x, y);
        cout << z << endl;
        return 0;
}
```

Stiva



main

Functii inline

- Am vazut ca apelul unei funcții presupune o serie de mecanisme complicate, fiind, pentru funcții scurte, mai eficient să inserăm codul corespunzător decât să facem un apel propriu-zis al funcției.
- Lucrul acesta se poate realiza prin precedarea funcției de cuvântul **inline**, informând, astfel, compilatorul că se dorește inserarea codului generat de funcție peste tot unde se face apelul ei.
- Cuvântul cheie **inline** se folosește doar la declararea funcției, nu și la apel. Multe compilatoare modifică, în mod implicit codul, generând cod inline atunci când consideră că pot, astfel, să optimizeze programul.

Exemplu:

```
inline int suma_numere(int a, int b) {   //definitie
        return a + b;
}

int main() {
        int rezultat{};
        rezultat = suma_numere(100, 200); //apel
        return 0;
}
```

Functii recursive

- O functie recursiva este o functie care se autopeleaza
  - Fie direct, fie indirect prin intermediul altei functii

- Daca apelam aceeasi functie de doua sau mai multe ori pentru a rezolva o problema, spunem ca rezolvam problema in mod recursiv
  - Cazul de baza
  - Cazul recursiv: impartirea restului problemei in subprobleme si apelarea recursiva

- Sunt multe probleme care se pot rezolva in mod recursiv

- Probleme matematice: calculul factorialului, generarea sirului lui Fibonacci, fractali, etc.

- Cautare si sortare

Functii recursive

- **Exemplu: Calculul factorialului**

0! = 1
n! = n * (n-1)!

Cazul de baza: factorial (0)=1
Cazul recursiv: factorial (n)=n * factorial(n-1)

```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;   //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(8) << endl; //40320
    return 0;
}
```

Functii recursive

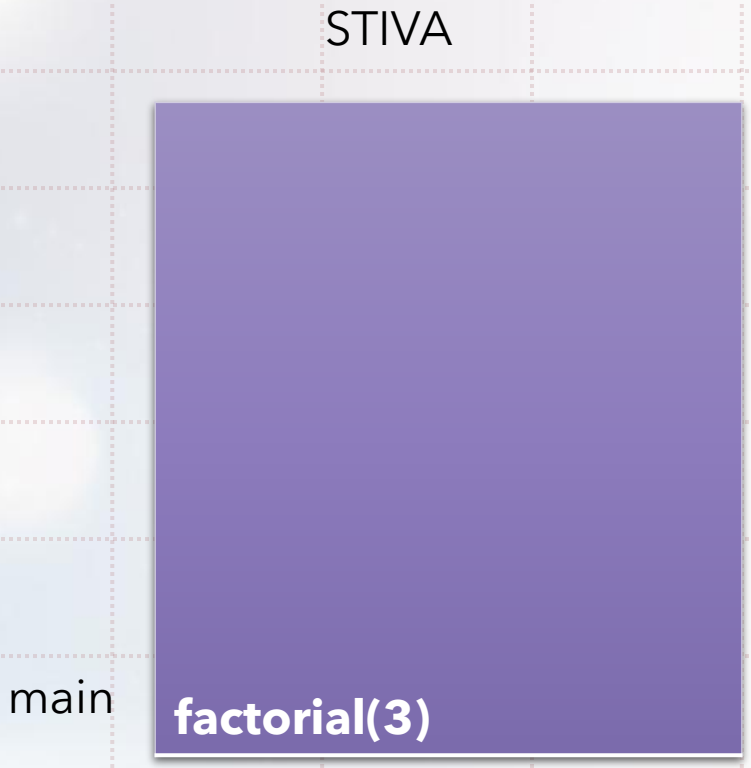
- **Exemplu: Sirul lui Fibonacci**
  - **Cazurile de baza:**
    Fib(0)=0
    Fib(1)=1
  - **Cazul recursiv:**
    Fib(n)=Fib(n-1)+Fib(n-2)

```
unsigned long fibonacci(unsigned long n) {
    if (n <= 1)
    return n; //cazurile de baza
    return fibonacci(n - 1) + fibonacci(n - 2);   //cazul recursiv
}

int main() {
    cout << fibonacci(30) << endl;   //832040
    return 0;
}
```

Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

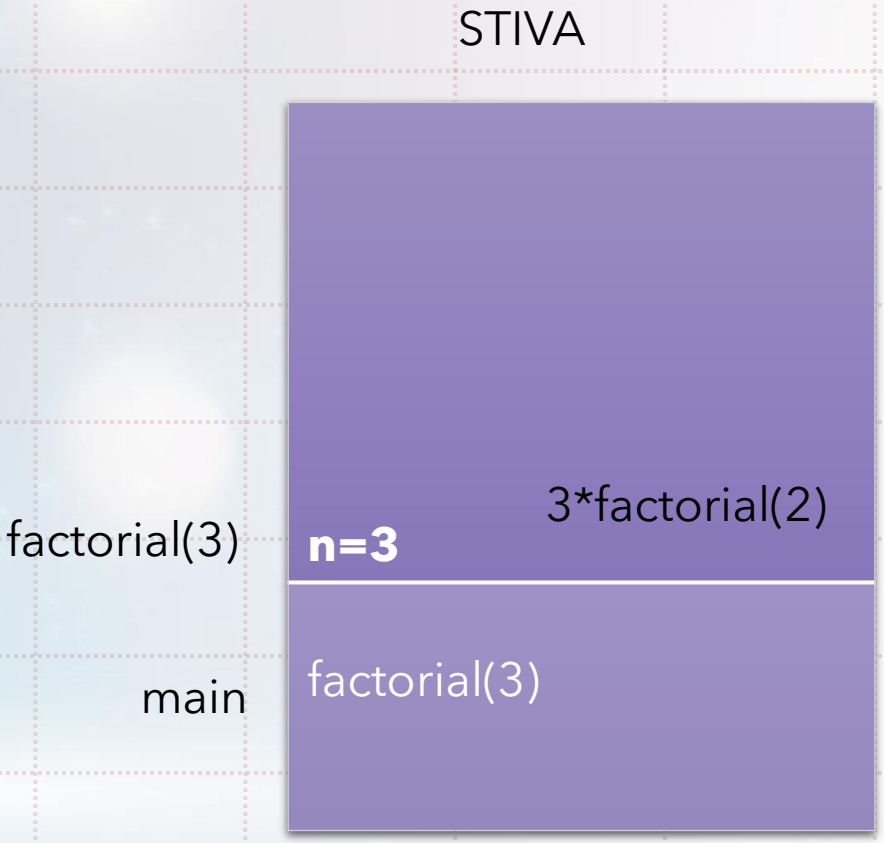
```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```



Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

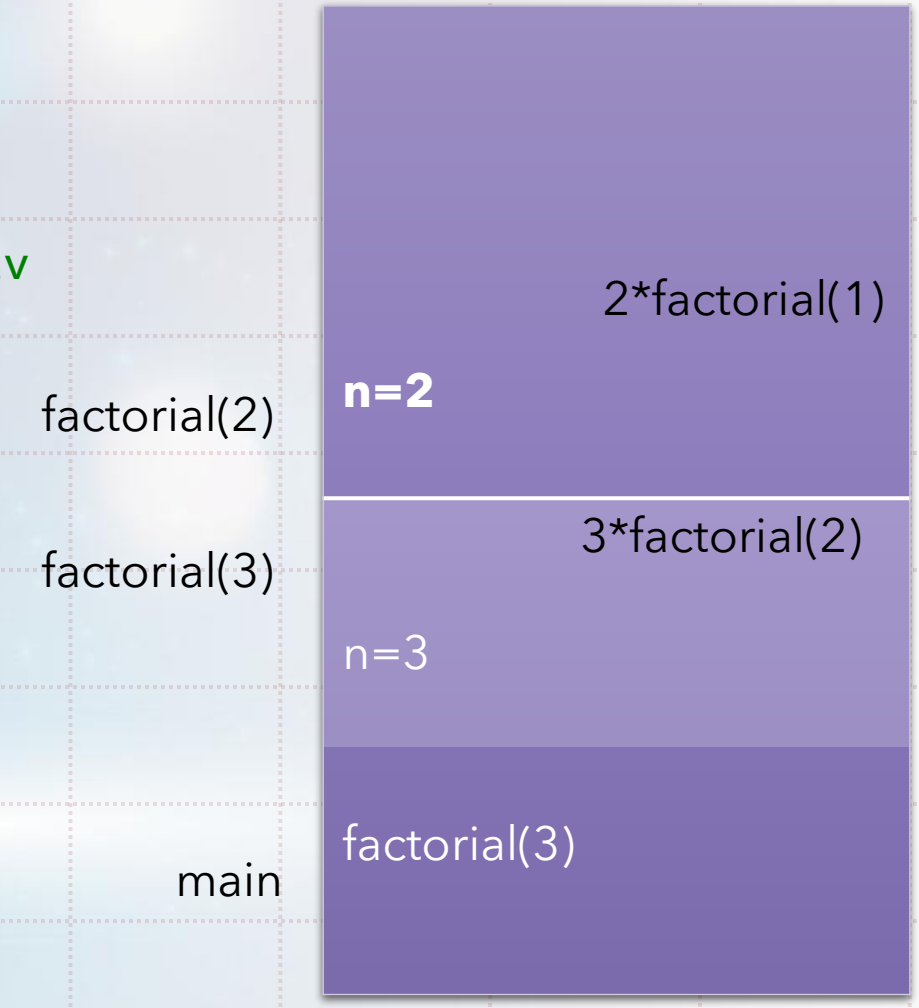
```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```



Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

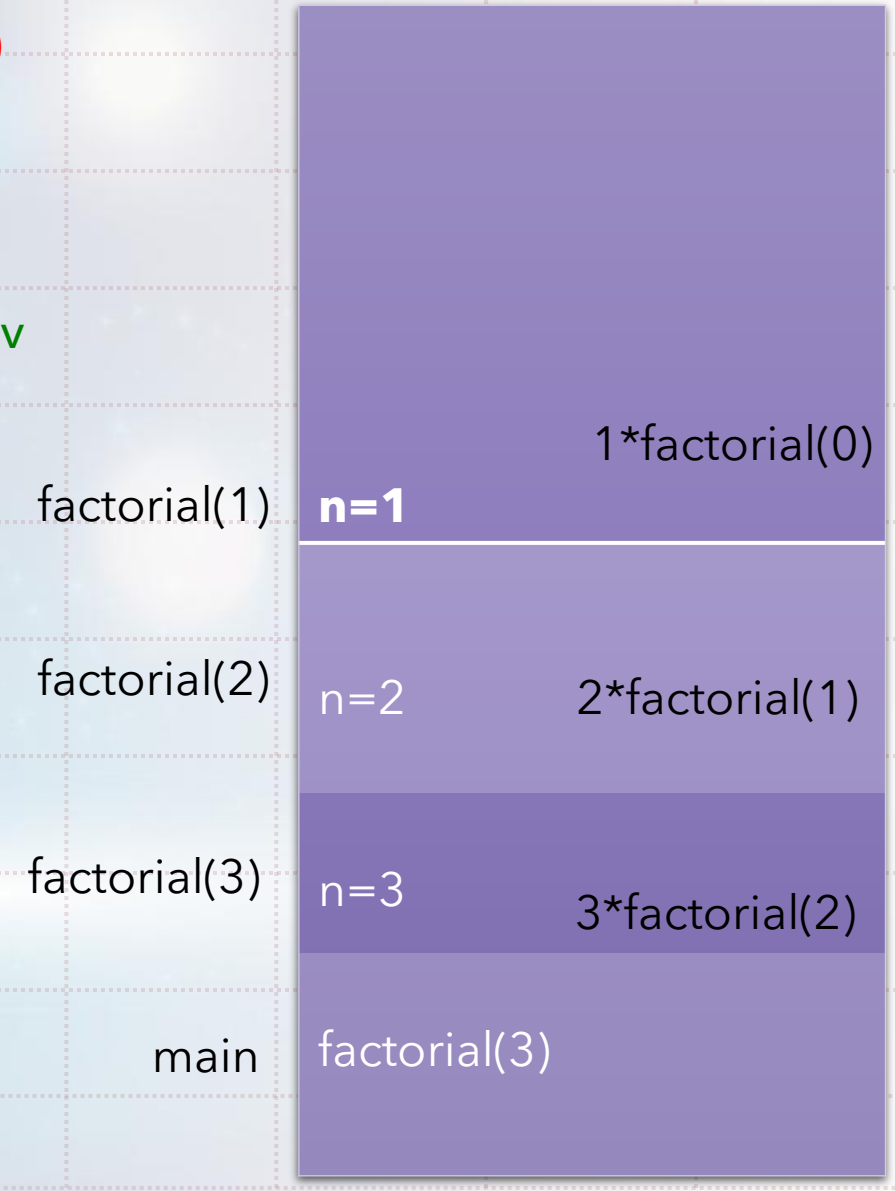
```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```



Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

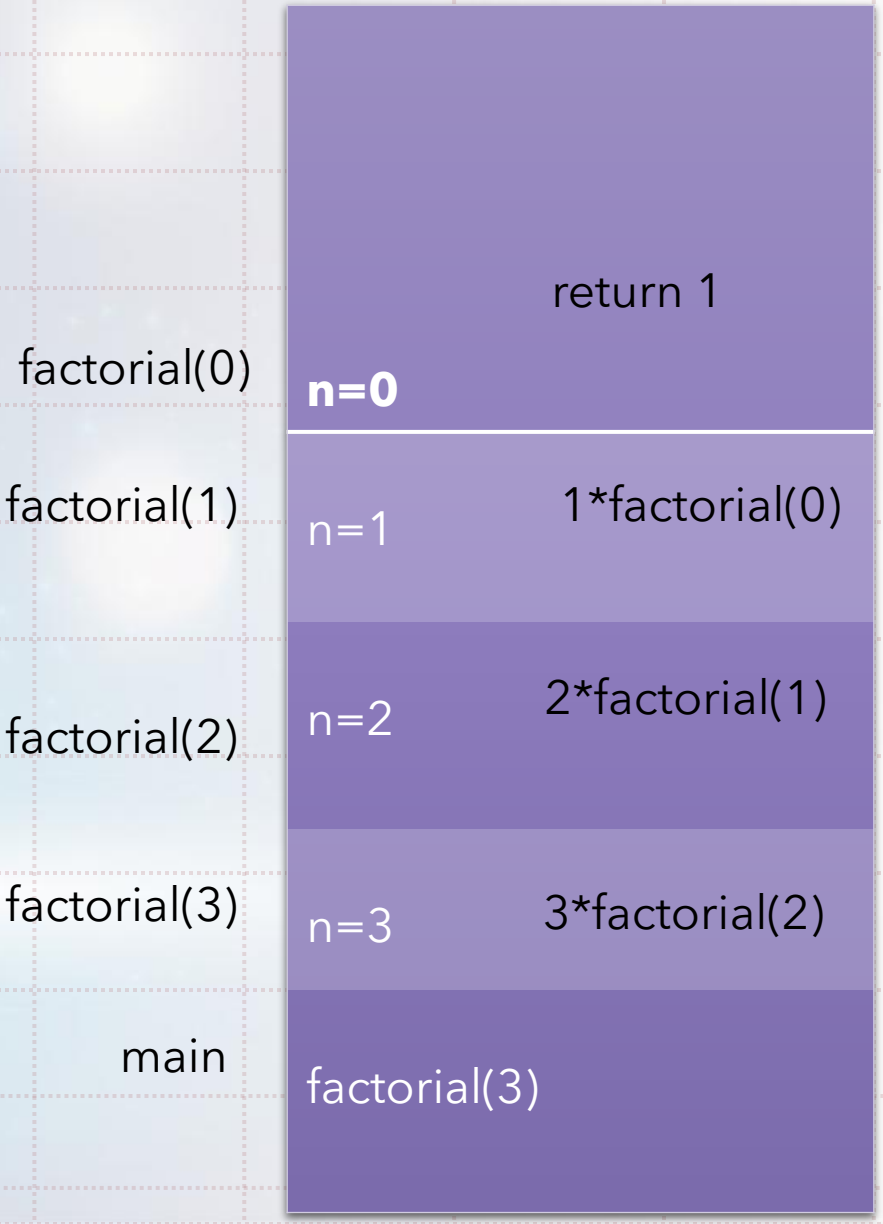
```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```



Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```



Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```

factorial(0)

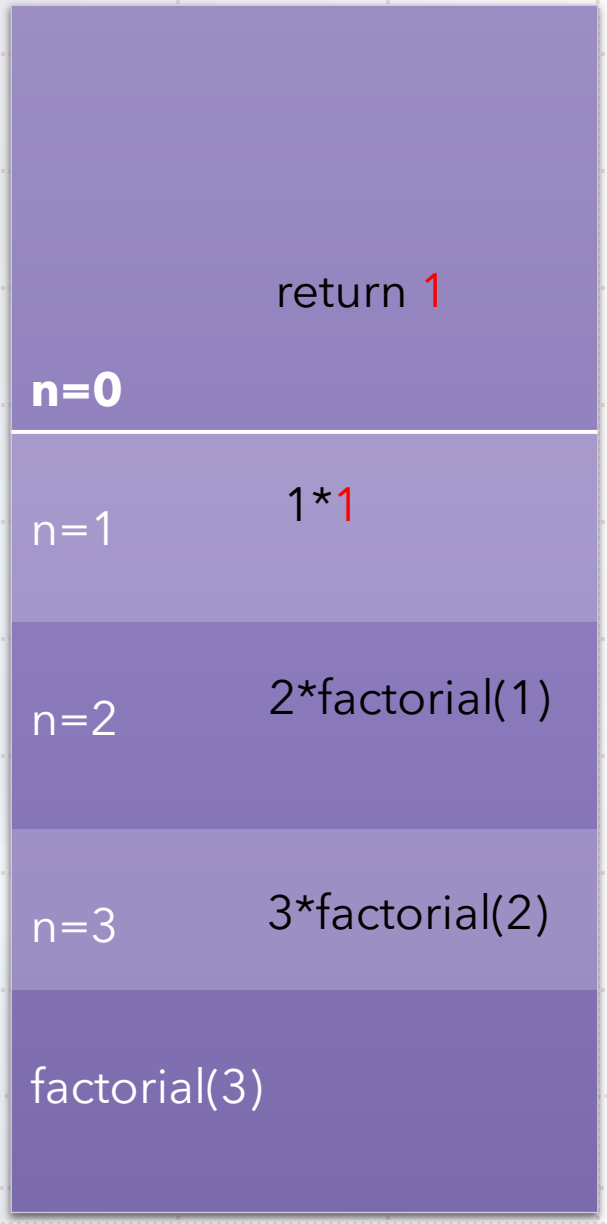
factorial(1)

factorial(2)

factorial(3)

main

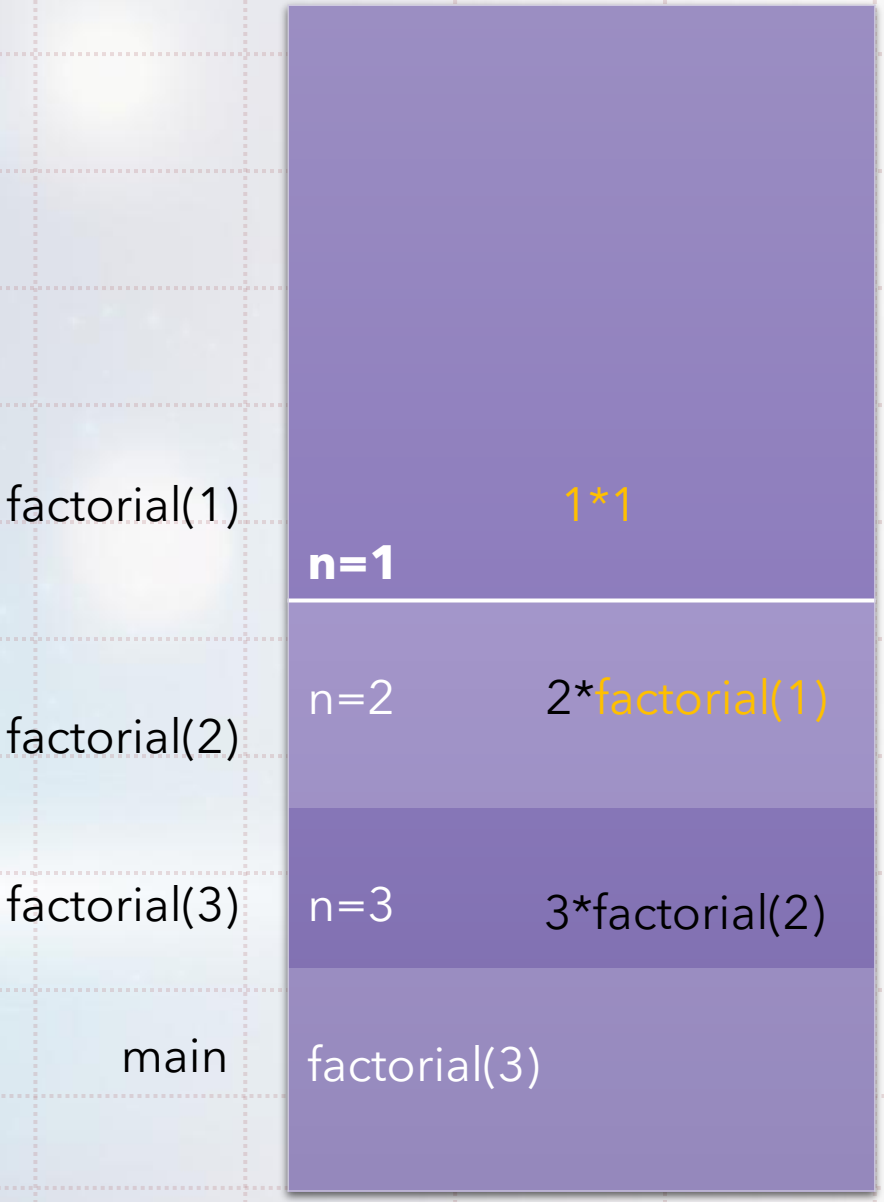
STIVA



Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

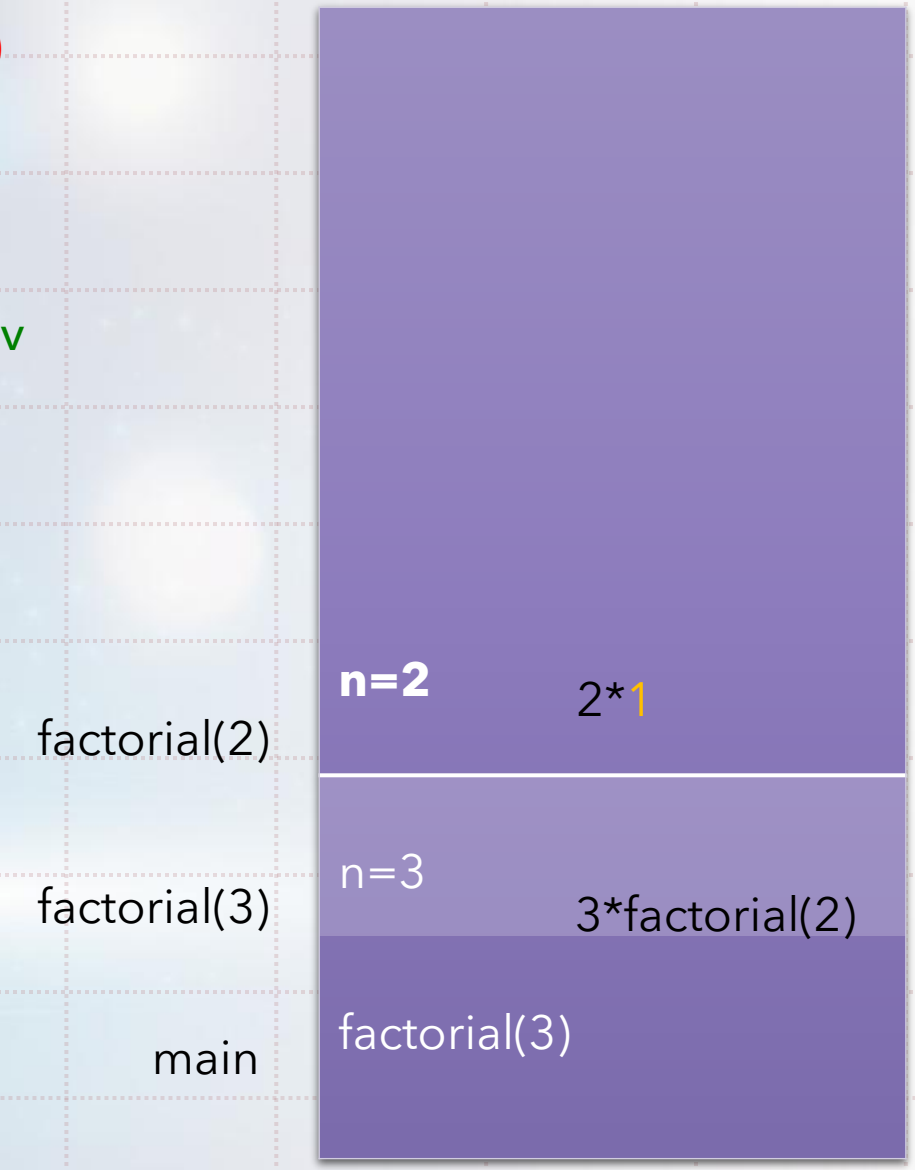
```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```



Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```



Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```

Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```

Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```

Functii recursive

- **Exemplu: Calculul factorialului (memorarea in stiva)**

```
unsigned long factorial(unsigned long n) {
    if (n == 0)
    return 1;    //cazul de baza
    return n * factorial(n - 1); //cazul recursiv
}

int main() {
    cout << factorial(3) << endl; //6
    return 0;
}
```

Functii recursive

- **Exemplu: Algoritmul de cautare binara (recursiv)**

```
void cautareBinaraRecursiv(int tab[], int st, int dr, int x) {
        int gasit = 0;
        int mijl = (st + dr) / 2;
        if (tab[mijl] == x)
                gasit = 1;
        else
                if (x < tab[mijl])
                        return cautareBinaraRecursiv(tab, st, mijl - 1, x);
                else
                        return cautareBinaraRecursiv(tab, mijl + 1, dr, x);
        if(st>dr)
                cout << "Nu s-a gasit elementul cautat !" << endl;
        else
                cout << "Elementul cautat apartine sirului !" << endl;
}
int main()
{……
 cautareBinaraRecursiv(tab, 0,n-1, x);
}
```

Functii recursive

Observatii importante!

- Recursivitatea este o forma de iteratie, deci tot ce se poate face recursiv se poate face si iterativ si viceversa

- Folositi recursivitatea doar atunci cand are sens (cand problema in cauza admite o solutie recursiva). De exemplu, nu folositi recursivitatea cand vreti sa numarati de la 1 la 10.

- Recursivitatea poate necesita multe resurse, in unele cazuri preferandu-se metoda iterativa (chiar daca este mai putin eleganta si uneori mai greu de inteles) in schimbul celei recursive.

- Nu uitati cazul/cazurile de baza
  - Cu ajutorul lor se termina recursivitatea
  - Daca nu opriti recursivitatea atunci veti avea o recursivitate infinita

https://test-master.space