

# C3: Introducere in programarea obiectuala

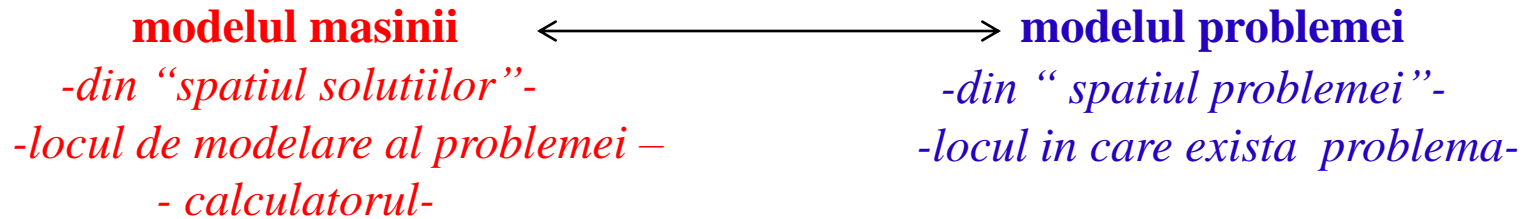
## Cuprins:

- POO – Introducere - Concepte de baza; Clasa si obiect
- Structura unei clase
- Functii membre ale unei clase
- Constructorii si destructorul unei clase
- Crearea si utilizarea obiectelor
- Cuvantul cheie const - continuare
- Cuvantul rezervat static -continuare

# C3: Introducere in programarea obiectuala

## Evolutia limbajelor de programare

Programele sunt traduceri. Programatorul stabileste asocieri :



Evolutie limbaje	vs.	Gradul de abstractizare al masinii pe care se ruleaza
Limbaj de asamblare		Scazut (problema e pusa in termenii masinii)
*Fortran, BASIC, C		Mediu (abstractizari ale lb. de asamblare)

*\*imbunatatire, dar, in prima instantă de abstractizare, programatorul trebuie sa gandeasca in termeni de structura a masinii de calcul mai degraba decat in termenii problemei.*

*=> programele sunt greu de implementat, scump de intretinut*

Alternativa la modelarea masinii este **modelarea problemei** -> lb. de nivel inalt

## Abordarea orientata pe obiecte



- pune la dispozitie unelte pentru reprezentarea elementelor din *spatiul problemei*
- reprezentare este suficient de *generală* -> programatorul nu e constrans la un anumit tip de problema
- ne referim, in acelasi timp, la *elementele din spatiul problemei* cat si la reprezentarea lor in *spatiul solutiei* prin **“obiecte”**.

*\*cu siguranta vor fi necesare si obiecte care nu au analog in spatiul problemei.*



**IDEEA** – programul se poate adapta la forma problemei prin adaugarea de *tipuri noi de date*, iar daca se citeste *codul ce descrie solutia* – se citesc cuvinte *ce descriu si problema de rezolvat=>*

**=> cea mai flexibila si puternica abstractizare de limbaj de pana acum**

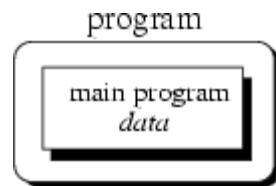


**POO permite descrierea solutiei in termenii problemei, mai repede decat in cei ai masinii pe care se ruleaza solutia.**

# C3: Introducere in programarea obiectuala

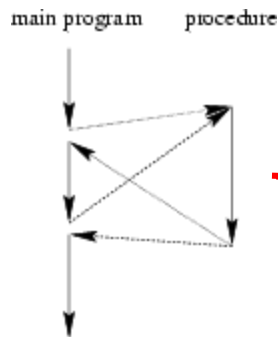
## Organizarea si functionarea aplicatiilor

### “Tipuri de programe”:



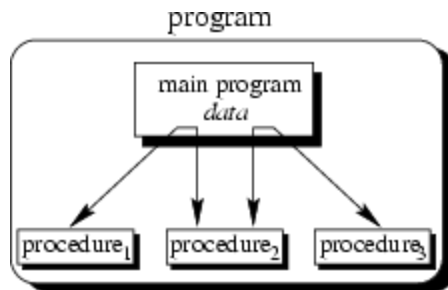
**Programare nestructurata:**

Programul principal opereaza direct cu datele (globale)



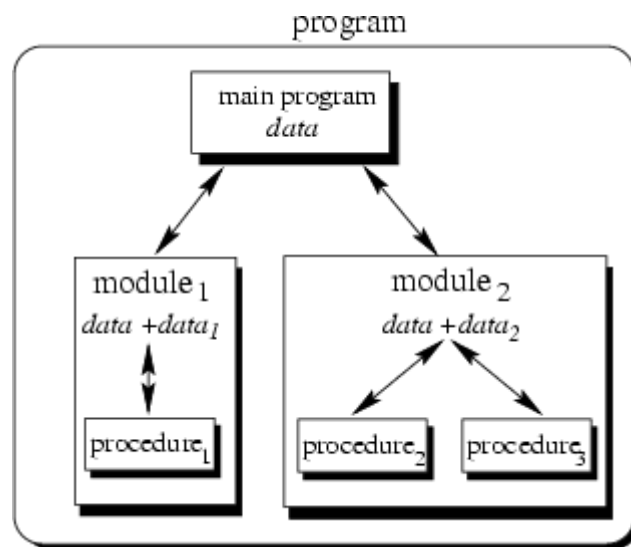
**Programare procedurala:**

Programul principal apeleaza la proceduri. Dupa apelul acestora, controlul revine programului principal.



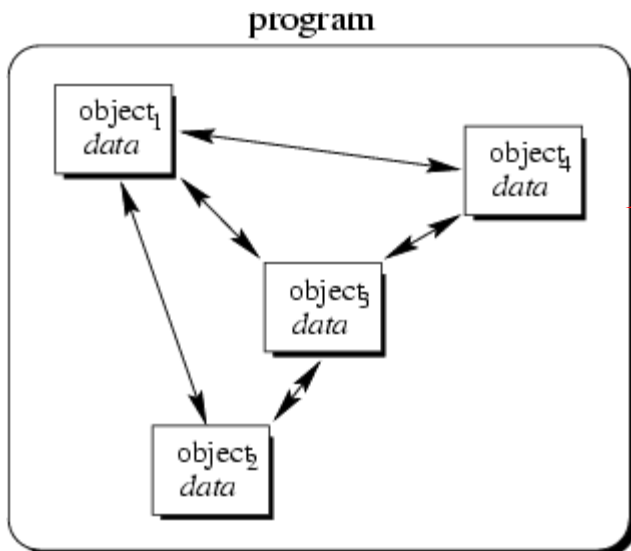
**Programare procedurala:**

Programul principal coordoneaza ce procedura este apelata si ce date sunt trimise ca parametri.



### Programare modulara:

Programul principal coordoneaza alegerea si lansarea in executie a modulelor adecvate catre care sunt trimise ca parametri datele corespunzatoare. Fiecare modul poate avea datele proprii.

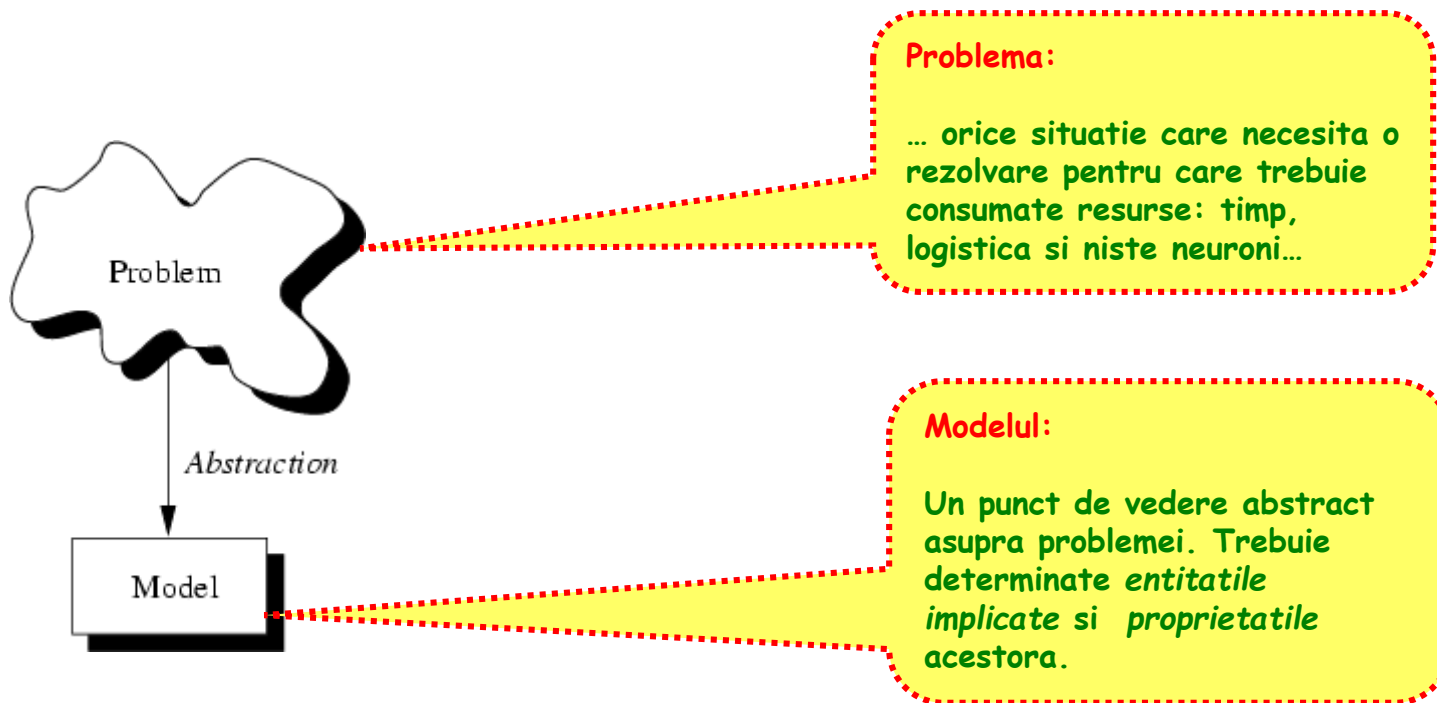


### Programare obiectuala:

Obiectele (entitatile) unui program au comportamente proprii si interactioneaza unele cu celelalte, schimband mesaje intre ele.

# C3: Introducere in programarea obiectuala

## Abordarea “problemelor”:



## Proprietatile entitatilor:

- *date – caracteristici/atribute ce sunt definatorii*
- *functii – actiuni/comportamente (care folosesc sau modifica datele)*

# C3: Introducere in programarea obiectuala

In cadrul programarii obiectuale (C++):

- se imbina structurile de date cu algoritmii care le prelucreaza / manipuleaza / utilizeaza rezultand tipuri de date complexe (definite de programator) - clase

=> **Avantaje:**

- Modularizare sporita a aplicatiei
- Usurinta in impartirea sarcinilor in echipa
- Suport pentru extinderea usoara a aplicatiei

*\* In loc de biblioteca de functii – biblioteca de clase*

## C3: Introducere in programarea obiectuala

Cele 4 principia de baza ale limbajelor orientate pe obiect:

- Incapsulare
- Abstractizare
- Mostenire
- Polimorfism

- a) **Incapsularea datelor** – datele sunt protejate (private). Manipularea lor se face prin intermediul functiilor membre.
- b) **Abstractizare** ~ organizarea datelor cu metodele lor aferente in clase
- c) **Mostenire** - se permite construirea unor clase derivate (fara a rescrie clasa de baza)
- d) **Polimorfism** - facilitatea ca intr-o ierarhie de clase obtinute prin mostenire, o functie cu acelasi nume sa aiba forme (implementari) diferite



# C3: Introducere in programarea obiectuala

## Structura (compozitia) unei clase

### Structura cu functii membre:

```
struct dreptunghi
{
    private:    int lun; //implicit publice
               int lat;  //campuri
    public:    void modif(int=0 , int=0);
               int get_lun() const;
               int get_lat() const;
};

//implementarea fctiilor membre
void dreptunghi::modif(int l1, int l2) {
    lun=l1; lat=l2;
}
int dreptunghi::get_lun() const {
    return lun;
}
int dreptunghi::get_lat() const {
    return lat;
}


//main
dreptunghi p; //o variabila de tip dreptunghi
p.modif(10,10); //p.modif(); //p.modif(2);
cout<<p.get_lun();
```

### Clasa:

```
class dreptunghi
{
    int lun; //implicit private
    int lat; //atribute/date
    public:  void modif(int=0 , int=0);
               int get_lun() const;
               int get_lat() const;
}; //interfata clasei Abstractizare ~ organizarea
//datelor cu metodele lor aferente in clase
//implementarea metodelor
void dreptunghi::modif(int l1, int l2) {
    lun=l1; lat=l2;
}
int dreptunghi::get_lun(void) const {
    return lun;
}
int dreptunghi::get_lat(void) const {
    return lat;
}

//main
dreptunghi p; //instantierea unui obiect
p.modif(10,10); //send a message/make a request
cout<<p.get_lun();
```

**Incapsularea datelor**  
(pot sa fie manipulate  
doar prin metodele clasei)



O **clasa** e un tip de date definit de utilizator (o matrita) cu

- un **nume** propriu
  - un set de **attribute** (date)
  - un set de comportamente/functionalitati - functii membre /**metode** (un set de functii asociate structurii de date)
- ❖ asemanator tipului de date de baza – int: datele sunt valori intregi si se pot incrementa, decrementa,...etc

**Diferenta:** programatorul defineste clasa astfel incat sa il ajute sa *rezolve problema* care ii e data – mai degraba decat sa fie fortat sa foloseasca niste tipuri de date predefinite.

Dupa ce a fost **declarata** si **implementata** o clasa, pot sa fie **instantiate/create** oricate obiecte/variabile de acel tip (cu date proprii); iar apoi folosite la rezolvarea problemei.

Un **obiect** este o instanta a unei clase (datele au valori, iar functionalitatile pot fi folosite).

## C3: Introducere in programarea obiectuala

### Funcțiile membre (metode) ale clasei:

- pot accesa atributele și funcțiile membre ale clasei (inclusiv pe cele private)
- pentru a implementa funcțiile membre sunt necesare referiri la datele membre ale clasei (atributele clasei), dar nu se face referire la un obiect anume
- la apelare, funcția este informată asupra identității obiectului asupra căruia va acționa prin transferul unui parametru implicit – referința la obiectul care face apelul.
- pentru apelarea lor folosim operatorii de selecție: (.) sau (->)

Ex: dreptunghi p;

`p.modif(1,1);` // la apelul funcției `modif` compilatorul testează dacă există o funcție cu //acest nume și această semnătură declarată și implementată pentru tipul de date //dreptunghi și, dacă da, se apelează funcția și i se transmite referința la p (adresa lui p)

- pot fi definite ca inline
- se pot supradefini
- în general, sunt publice, dar pot fi și private

Când ar avea sens să declar o metodă privată?

# Separarea interfetei de implementarea efectiva

//interfata - intr-un fisier **header** cu numele clasei **point** – si cu extensia **.h**

```
class point                                // class – cuvant rezervat; point – numele clasei
{                                           // private – cuvant rezervat; atr. vizibile doar in interiorul clasei
    private:                               // atribute, membrii de tip date ai clasei; poate sa lipseasca
        int x, y;                          // public – cuvant rezervat; vizibile oriunde
    public:                                // declararea functiilor membre ale clasei
        void modifica(int =0, int =0);     // daca vreau pot face implementarea aici
        int getx() const;                  // dar e preferabil sa fie facuta separat
        int gety() const;
};
```

//implementarea - intr-un fisier cu nume clasei **point** si cu extensia **.cpp**

```
#include "point.h"                        // trebuie inclus headerul deoarece ne referim la acea zona de cod
// implementarea metodelor                // includerea se face folosind ""
void point::modifica(int vx, int vy){     // se foloseste operatorul de rezolutie pentru a preciza ca e
    x = vx;                               // vorba despre metoda modifica din clasa point
    y = vy;                               // valorile param. default nu mai sunt respecificate
}
int point::getx() const{                  //const - functia impl. nu modifica obiectul care face apelul
    return x;                             //const nu poate sa lipseasca in zona de impl.; functia nu ar mai
}                                          //fi recunoscuta
int point::gety() const{
    return y;
}
```

//aplicatia ce foloseste tipul de date point - intr-un fisier cu orice nume si extensia .cpp

```
#include "point.h"           // trebuie inclus deoarece o sa ne referim la tipul de date point
#include <iostream>           // headerele definite de programator se includ intre ghilimele
using namespace std;
```

```
int main()
```

```
{  point p, *p1, *vect;
```

// declarare si instantiere obiecte

// cine si cum alocata spatiu pentru p?

```
    p.modifica(1,1);
```

```
    cout<<p.getx()<<" "<<p.gety()<<endl<<endl;
```

// cum se apeleaza metodele?

```
    p1=new point;
```

// alocare spatiu

```
    p1->modifica(2,2);
```

```
    cout<<p1->getx()<<" "<<p1->gety()<<endl<<endl;
```

```
    int n; cin>>n;
```

```
    vect=new point[n];
```

// alocare spatiu

```
    for (int i=0;i<n;i++)
```

```
        vect[i].modifica(i,i);
```

```
    for (int i=0;i<n;i++)
```

```
        cout<<vect[i].getx()<<" "<<vect[i].gety()<<endl;
```

```
    return 0;
```

```
}
```

// ce se intampla daca declaram x, y public?

// puteam sa ii accesam si in afara clasei:

// ex. cout<<p.x; **DAR se strica incapsularea**

Impartirea in fisiere header si sursa este necesara pentru usurinta in citirea codului (lizibilitate) si compilarea aplicatiilor mari.



Nu este insa obligatorie, dar e indicata.

Codul din cele 3 fisiere putea sa fie implementat intr-unul singur.

# C3: Introducere in programarea obiectuala

## Cuvantul cheie `this` (autoreferinta)

- ❖ Exista situatii cand, in cadrul implementarii unei functii membre, avem nevoie de adresa obiectului asupra caruia facem modificari/interogari.
- Ne putem referi la acest pointer folosind cuvantul cheie **this**. El este:
  - declarat implicit in orice metoda
  - initializat sa pointeze catre obiectul pentru care e invocata functia membru.

Principala utilizare este la implementarea de functii care manipuleaza direct pointeri (vom vedea mai departe).

**Alta utilizare:**

```
void point::modifica(int x, int yy)
{
```

```
    this->x = x; //this->x se refera la atributul x al obiectului cu care se lucreaza
                // x este parametrul functiei
    this->y = yy; // puteam sa scriu si y=yy; nu aparea nicio confuzie
```

```
}
```

```
point p1,p2;
p1.modifica(1,1);
p2.modifica(1,1);
```

# C3: Introducere in programarea obiectuala

## Constructorul (constructorii) si destructorul unei clase

- in cazul variabilelor de tip de date de baza, compilatorul este cel care alocă spatiu de memorie si eventual face initializarea acelei variabile
- nu acelasi lucru se intampla in cazul variabilelor de tip definit de utilizator – compilatorul nu dispune de metode de initializare suficient de complexe.

Considerati exemplul urmator – clasa vector de intregi de dimensiune n:

```
class vector {  
    // private:  
        int n;                // dimensiune  
        int *vec;             // vectorul de intregi  
    public:    //.....alte metode  
        int getlung(void) {  
            return n;  
        }  
        int get_elm(int i){  
            return vec[i];  
        }  
};  
  
//utilizare  
vector v;                //cat spatiu se alocă automat? Ce as dori sa se intample?
```

## Constructorii

- sunt functii speciale, membre ale clasei (in general, declarate public);
- se ocupa de crearea, initializarea sau realizarea de copii ale obiectelor;
- au acelasi nume ca si clasa;
- NU au tip de date returnat;
- au rolul de a crea obiecte de tipul clasei;
- se pot supradefini;
- nu se apeleaza explicit: ~~point p; p.point(2,2);~~ point p(2,2);
- se apeleaza doar cand e declarant/creat obiectul point x;

Ar trebui implementati de programator.

Daca nu se gaseste nicio implementare efectiva de constructor, compilatorul va genera automat doi constructori default (standard C98).

**Primul:** - constructor fara parametri care se ocupa cu alocarea de spatiu pentru datele membre ale clasei.

Ex de apel: vector v; //aici se apeleaza un constructor fara parametri;  
//deoarece clasa vector nu are implementat un astfel de constructor, compilatorul  
//genereaza automat unul care alocu spatiu pentru un int si un pointer la int.



```

//vector.h
class vector {
    // private:
        int n;                // dimensiune
        int *vec;              // vectorul de intregi
    public:
        vector(int ); //constructor cu un parametru- dimensiunea vectorului
        vector(int, int*); //constructor cu 2 parametri- dimensiunea si valorile vectorului
        void copie(const vector &); // si alte functii membre
        int getdim();
        int* getvec();
        void set_elm(int, int);
        int get_elm(int);
        void afis();
};

```

```
#include "vector.h"
#include <cstdlib>
#include <iostream>
#include <assert.h>
using namespace std;
```

```
vector::vector(int n)    // constructor
{
    this->n = n;
    if (n>0)  vec = new  int[n];
              else vec=NULL;
}
```

```
vector::vector(int n, int *a) // constructor
{  this->n = n;
   if (n>0 && a!=NULL){
       vec = new  int[n];
       for(int i = 0; i < n; i ++){
           vec[i] = a[i];
       }else vec=NULL;
   }
```

```
void vector::copie(const vector &v)
{
    this->n = v.n;
    if (vec != NULL) delete [] vec;
    vec = new  int[n];
    for(int i = 0; i < n; i ++){
        vec[i] = v.vec[i];
    }//mai trebuie testat ceva?
```

**//alocarea de spatiu + testele aferente s-ar putea muta**  
**//intr-o functie (private) si apela acolo unde e nevoie**  
**// - >Tema**

```
int* vector::getvec()
{
    return vec;
}
```

```
int vector::getdim()
{
    return n;
}
```

```
void vector::set_elm(int i, int val)
{
    vec[i]=val;
}
```

```
int vector::get_elm(int i)
{
    return vec[i]; //ar trebui un test i<n; i>-1
}
```

```
void vector::afis()
{
    cout<<"dim este: "<<n<<" si elem. sunt: ";
    for (int i=0;i<n;i++)
        cout<<vec[i]<<" ";
    cout<<endl;
}
```

```
#include "vector.h"
#include <iostream>
using namespace std;
```

```
int main(int argc, char *argv[])
{ vector v(3); //creez un obiect de tip vector cu 3 elem. si spatiu alocat – prin apel constructor cu un param.
               //cum arata memoria?
  for (int i=0;i<v.getdim();i++)
    v.set_elm(i,i);
  for (int i=0;i<v.getdim();i++)
    cout<<v.get_elm(i)<<" ";
  cout<<endl<<"_____ "<<endl;
```

**// vector px; //pot sa creez un obiect asa?**

**//ce se intampla de fapt?**

**//daca implementez un constructor, nu se mai genereaza automat cel default**

```
vector p(v.getdim()); //era bine sa am constructorul fara parametrii implementat
p.copie(v); // dar p=v – reprezentati in mem
p.afis();
cout<<endl<<"_____ "<<endl;
```

```
vector p1(v.getdim(),v.getvec()); //creez un obiect de tip vector cu lungime 3 si se alocata spatiu si se copiaza val
                                   //din vectorul v; se foloseste constructorul cu 2 param.
p1.afis();
cout<<endl<<"_____ "<<endl;
```

```
vector p2(v); //exista si un constructor de copiere default; el pentru cine va alocata spatiu?
v.set_elm(0, 1000);
p2.afis(); //Ce se afisaza? Care este problema?
return 0;}
```

# C3: Introducere in programarea obiectuala

## Constructorul de copiere

- alta categorie de constructor
- necesitatea sa apare in contextul realizarii de copii in primul rand pentru:
  - parametrii transmisi in functii prin valoare
  - la returnarea unui obiect dintr-o functie
  - la crearea unui obiect temporar
- daca nu e definit de programator - se genereaza unul default care initializeaza datele noului obiect copiind bit cu bit fiecare atribut din sursa in fiecare atribut din destinatie
- nu este o solutie daca am attribute variabile dinamice – trebuie facuta implementarea de catre programator – altfel vom avea niste copii superficiale(**shallow copy**) - > erori logice.

Parametrul – obiectul caruia ii facem copia trebuie sa fie transmis prin referinta

**De ce?**

Altfel – daca e transmis prin valoare cineva (?) ar trebui sa ii faca o copie – dar de-abia acum e implementata aceasta functie (constructorul de copiere)

```
class vector
{
    ...
    public:
        vector (const vector &);
    ...
};
```

```
vector::vector(const vector &v)
```

```
{
    n = v.n;
    vec = new int [n];
    for (int i = 0; i < n; i ++)
        vec[i] = v.vec[i];
}
```

// sau n = v.getdim(); este n>0?  
//tb sa verific si ca v.vec!=NULL  
// tema!!  
//**Atentie!** Nu incercati sa eliberati spatiul  
//pt vec (delete[] vec;) inainte de alocare.  
// De ce?

```
//apel:
vector a(3);
vector b(a);
```

//am mai vazut vreodata ceva similar?  
//int \*x=new int(3); int y(4);

```

struct Carte {
    char * titlu;
    char * autor;
    int id;

```

Inca un exemplu:

```

public:

```

```

    Carte(){
        titlu=autor=NULL;
        id=0;
    }

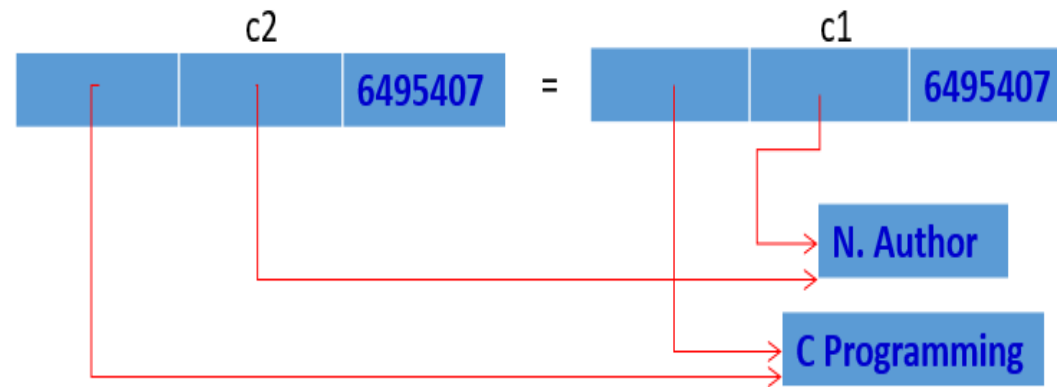
```

As putea sa scriu un singur constructor in loc de cei 2 din exemplu? Cum?

```

    Carte(char*t, char*a, int i){
        if (t!=NULL) {titlu=new char[strlen(t)+1]; //in heap
                        strcpy(titlu,t);
        }else titlu=NULL;
        if (a!=NULL){ autor=new char[strlen(a)+1]; //in heap
                        strcpy(autor,a);
        }else autor=NULL;
        id=i;
    }
};

```



```

//main

```

```

Carte c1("N. Author","C Programming",6495407), c2; //c1, c2 pe stack

```

```

c2=c1;    //ce se intampla daca modific c2.nume?

```

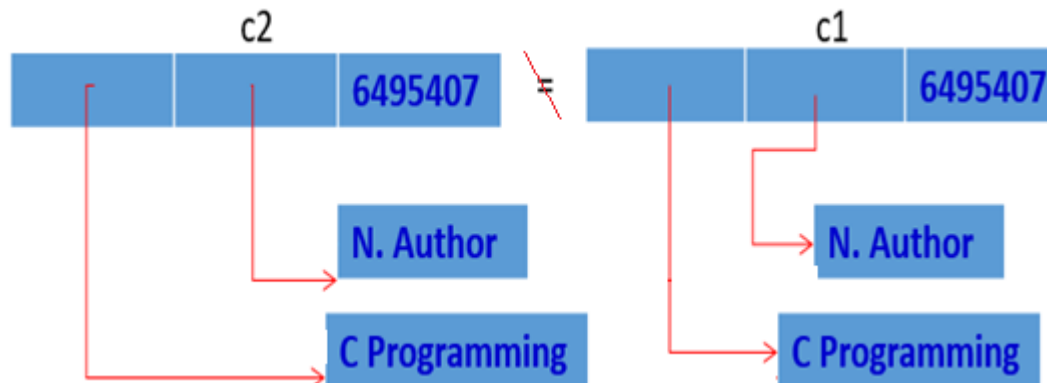
Copii profunde – se realizeaza- la creare obiectului – constructor de copiere sau mai tarziu in program – fct. modifica/copiaza

//in clasa Carte

```
Carte(const Carte& c){  
    if (c.titlu!=NULL) {titlu=new char[strlen(c.titlu)+1]; //in heap  
                        strcpy(titlu, c.titlu);  
    }else titlu=NULL;  
    if (c.autor!=NULL){ autor=new char[strlen(c.autor)+1]; //in heap  
                        strcpy(autor, c.autor);  
    }else autor=NULL;  
    id=i;  
}
```

//in main

```
Carte c1("N. Author","C Programming",6495407), c2(c1); //c1, c2 pe stack
```



```

//fctie membra a clasei Carte
void copiaza (const Carte& c){
    if (titlu!=NULL) delete[] titlu; // altfel memory leak
    if (c.titlu!=NULL) {titlu=new char[strlen(c.titlu)+1]; //in heap
                        strcpy(titlu, c.titlu);
    }else titlu=NULL;
    if (autor!=NULL) delete [] autor; // altfel memory leak
    if (c.autor!=NULL){ autor=new char[strlen(c.autor)+1]; //in heap
                       strcpy(autor, c.autor);
    }else autor=NULL;
    id=i;
}

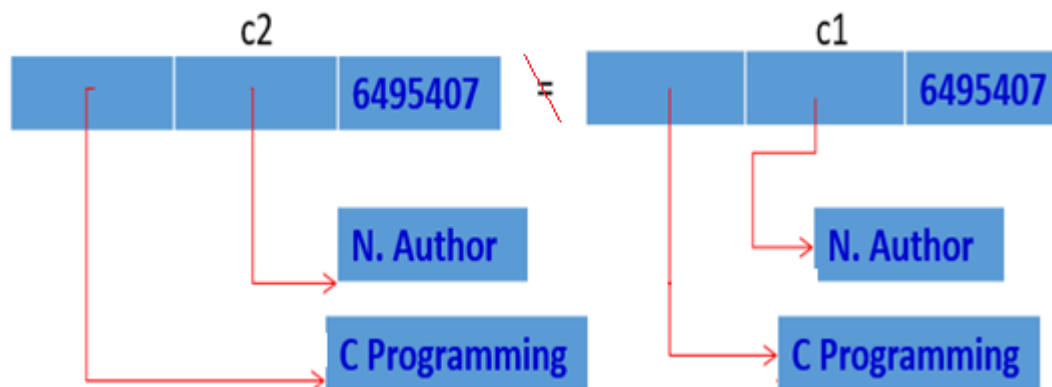
//main
Carte c1("N. Author","C Programming",6495407), c2; //c1, c2 pe stack
c2.copiaza(c1);

```

1. Ce se intampla daca in constructorul fara parametrii nu faceam titlul si autorul NULL?
2. De ce nu eliberam spatiul pentru titlu si autor si in constructorul de copiere?

R1 – Am fi avut eroare aici. Titlu nu e nici NULL, nici nu i-a fost alocat spatiu – dam delete pe o zona de memorie nealocata.

R2 – fiind intr-un constructor, titlu si autor inca nu au primit niciun spatiu de memorie (nici nu sunt NULL). Nu avem ce elibera.





## C3: Introducere in programarea obiectuala

### Destructorul:

- `~nume_clasa ();`
- fara tip sau parametri
- are rolul de a elibera spatiul de memorie ocupat de un obiect
- daca nu implementez unul - se genereaza automat un destructor ( functie publica)

**Atentie:** Pentru clasele cu atribute `tip_date *` - implementati destructorul! Voi alocati spatiu -> voi il eliberati!

Are urmatoarea implementare pentru clasa vector:

```
vector::~~vector(){  
    if (vec!=NULL) delete [ ] vec;  
}
```

Se apeleaza de catre utilizator (explicit) doar in contextul pointerilor:

```
vector *v=new vector(3); // un pointer catre un obiect de tip vector de dimensiune 3  
delete v; //aici am de fapt apel destructor
```

Altfel se apeleaza automat cand o variabila elibereaza spatiul de memorie (la finalul duratei sale de viata).

## Inca un exemplu:

```
//pers.h
```

```
#include <cstdlib> //pentru NULL
```

```
class pers
```

```
{
```

```
    char*nume;
```

```
    int varsta;
```

```
public:
```

```
    pers(char* n=NULL, int v=0);
```

```
    pers(const pers&);    //trebuie implementat – pentru ca am un atribut de tip char*
```

```
    ~pers();              //trebuie implementat – pentru ca am un atribut de tip char*
```

```
    void afisare();
```

```
};
```

```

//pers.cpp
#include "pers.h"
#include <iostream>
#include <string.h>
using namespace std;

    pers::pers(char* n,int v)
//nu mai precizez inca o data valorile
//pentru param default
    {
        if (n==NULL) nume=NULL;
        else {
            nume=new char [strlen(n)+1];
            strcpy (nume,n);
        }
        varsta=v;
    }

pers::pers(const pers& p)
{
    if (p.nume==NULL) nume=NULL;
    else {
        nume=new char [strlen(p.nume)+1];
        strcpy (nume,p.nume);
    }
    varsta=p.varsta;
}

```

```

pers::~pers()
{
    if (nume!=NULL) delete [] nume;
}

void pers::afisare()
{
    if (nume==NULL)
        cout<<"John Doe"<<varsta<<endl;
    else
        cout<<nume<<" are varsta "<<
            varsta<<endl;
}

```

Declarati obiecte de tip pers!

Tema: Implememtati o functie care seteaza numele cu o noua valoare (faceti toate testele necesare). Utilizati fctia in cei 2 constructori. Fctia poate sa fie private.

```

#include <cstdlib>
#include <iostream>
#include "pers.h"
using namespace std;

int main(int argc, char *argv[])
{
    pers p;    // apel constructor cu parametri default NULL si 0
    pers p1("ana"); // apel constructor cu nume- "ana" si parametru default 0 pt varsta
    pers p2("maria",19); //ce sunt "maria", "ana", 19?
    p.afisare();
    p1.afisare();
    p2.afisare();

    pers p3(p2); // apel constructor de copiere
    p3.afisare();
    pers p4(pers("ceva")); // ce se intampla aici? Ce e pers("ceva") ?
                        //rezulta un obiect temporar => are tipul const pers
    return 0; //este neaparata nevoie ca param constr.copiere sa fie ref. const.
}

```

Cand este apelat destructorul din pers si de cate ori?

Cum creez dinamic un vector de tip pers? Cine se apeleaza pt alocarea de spatiu?

Ce se intampla daca nu am constructor fara parametri, ci doar unul cu parametri?

//pers \*v = new pers[3]; // **se apeleaza** de 3 ori **constructorul fara param.** – in cazul

//nostru cel cu parm. default; "Tema" - testati

# C3: Introducere in programarea obiectuala

## Ambiguitati la apelul constructorilor cu parametri cu valori implicite

```
class point{
    int x,y;
    public:
        point();           // c1
        point(int = 0);    // c2
        point(int, int = 0); // c3
};

point :: point()           //c1
{
    ...
}

point :: point(int nn) //nu trec din nou valoarea default //c2
{
    ...
}

point :: point(int nn1, int nn2) //c3
{
    ...
}
```

```
int main()
{
    nr n1;           // ambiguu c1 sau c2
    nr n2(1);        // ambiguu c2 sau c3
    return 0;
}
```

# C3: Introducere in programarea obiectuala

## Exemple creare obiecte:

```
class ora_exacta {
    int ora, minut, secunda;
public:
    ora_exacta (int, int, int );//parametri pt ora, minut, secunda
    ora_exacta (int, int ); //parametri pt ora, minut
    ora_exacta (int ); //parametri pt ora
    ora_exacta ( ) {secunda = minut = ora = 0;};
    ora_exacta (const ora_exacta &);
    ~ora_exacta (){};
};

ora_exacta::ora_exacta (int o, int m, int s){
    ora = o;
    minut = m;
    secunda = s;
}

ora_exacta::ora_exacta (int o, int m){
    ora = o;
    minut = m;
    secunda = 0;
}
```

```

ora_exacta:: ora_exacta (int o){
    ora = o;
    minut = 0;
    secunda = 0;
}

```

```

ora_exacta:: ora_exacta (const ora_exacta &o){
    ora = o.ora;
    minut = o.minut;
    secunda = o.secunda;
}

```

Am nevoie de constructorul de copiere?

**NU**

**//apel - main**

```

ora_exacta o1, o2(10), o3(12, 15), o4(3, 4, 5);
ora_exacta *po=new ora_exacta;
ora_exacta *po1=new ora_exacta(2);
ora_exacta *po2=new ora_exacta(2,2);
ora_exacta *po3=new ora_exacta(2,2,2);
ora_exacta *vec=new ora_exacta[10]; //se apeleaza constructorul fara param
//daca nu era implementat =>ERROR

delete po;
delete po1;
delete po2;
delete po3;
delete [] vec;

```

**//Cum puteam sa reduc dimensiunea codului, fara sa afectez functionalitatea sa?**  
**//ora\_exacta(int o=0, int m=0, int s=0);**

# C3: Introducere in programarea obiectuala

Observatii importante si rafinari ale ideilor prezentate anterior:

Trebuie remarcate asemanarile si diferentele dintre structuri si clase:



- a) - **asemanare fundamentala:** ambele pot contine atat date cat si functii
- **diferenta fundamentala:** pentru clase - membrii acestora sunt **implicit private**, spre deosebire de structuri unde acestia sunt **implicit public**;

Cuvintele cheie (modificatori de acces) ce modifica “vizibilitatea” membrilor:

- ***private*** (vizibil doar in zona de cod –clasa din care face parte)
- ***public*** (vizibil de oriunde)



- b) O clasa in C++ poate contine doua categorii speciale de functii
- constructori (dintre care doar unul de copiere)
- destructor (doar unul).



- c) **Functia membru – constructor** - are prin definitie acelasi nume cu cel al clasei si nu returneaza nimic.
- este apelata la construirea unui nou obiect de tipul respectiv.
  - exista macar doi pentru fiecare clasa (cel fara parametri si cel de copiere); daca nu ii implementez - sunt generati automat

d) Obiectele pot fi instantiate:

- **local** – la nivelul unor functii;
- **global** – avand statut de variabile globale.

Cand un obiect este **variabila globala**, constructorul este apelat atunci cand programul isi incepe executia. A se nota ca apelarea constructorului este facuta inainte de a lansa functia **main()** .

Cand un obiect este declarat ca **variabila statica locala**, apelarea constructorului se face la **primul apel al functiei** in care este declarat obiectul respectiv.

Cand un obiect este o **variabila locala (ne-statica)** a unei functii, constructorul este apelat de fiecare data cand functia este apelata, in momentul (locul) in care variabila este definita.



*e)* Destructorul este cea de-a doua functie speciala a unei clase.  
Este apelat atunci cand obiectul inceteaza sa mai existe.

Pentru obiectele care sunt declarate ca **variabile locale ne-stactice** destructorul este apelat (chemat) atunci **cand blocul in care este instantiat obiectul este parazit.**

Pentru **variabilele statice sau globale** destructorul este apelat **inainte de terminarea programului.**

**Atentie:** cand un program este intrerupt prin utilizarea unei secvente **exit()** destructorii sunt chemati doar pentu obiectele globale existente in momentul respectiv.

Pentru obiectele definite local, in acesta situatie de fortare a terminarii, nu se face apelul!

## Exemplu (ilustrativ pentru ordinea de apel a constructorilor si destructorilor):

```
#include <iostream>
using namespace std;
class Test{
public:
    Test() { // constructor
        cout<< "apelare constructor al clasei Test\n"; }

    Test(const Test &x) { // constructor copiere
        cout<< endl<<"apelare constructor de copiere al clasei Test"<<endl; }

    ~Test(){ // destructor
        cout<<"apel destructor"; }
};

// alte functii – in afara clasei Test
void func()
{
    Test l;    // variabila locala
    cout << "mesaj din functia func()" << endl;
}

void func(Test t)    //parametru transmis prin valoare
{
    cout << "mesaj din functia func(Test t)" << endl;
}
```

```
Test g; //obiect declarat global

int main(){

    cout << "mesaj din main()" << endl;

    Test x;    // obiect local in main()

    func();

    func(x);

    return 0;
}
```

## Mesajele in urma rularii programului demonstrativ

- apelare constructor al clasei Test - pt. obiectul global g
- mesaj din main()
- apelare constructor al clasei Test - pt. obiectul x din main()
- apelare constructor al clasei Test - pt. obiectul l din func()
- mesaj din functia func()
- apel destructor - pt. eliberarea spatiului ocupat de obiectul din func()
- apelare constructor de copiere al clasei Test - pentru transmitere prin valoare a param. in fctiei.
- in functia func(Test t )
- apel destructor - pt. elib. sp. ocupat de copia param din func(test)
- apel destructor - pt. eliberarea spatiului ocupat de obiectul din main()
- apel destructor - pt. eliberarea spatiului ocupat de obiectul global

## C3: Introducere in programarea obiectuala

Functii membru - const, referinte constante si obiecte constante :



Cuvantul cheie (rezervat) **const** apare uneori **dupa lista de argumente a unor metode** pentru a specifica faptul ca functia membra respectiva **nu modifica (nu altereaza)** attributele obiectului care o apeleaza, doar le citeste sau utilizeza.

**Referinta/variabila constanta** – inseamna ca variabila/parametrul de tip referinta nu are voie sa isi schimbe valoarea in functia in care este transmis (orice incercare -> ERROR).

Exemplu:

```
class dreptunghi
{
    private:                // nu este necesar
        int lun;
        int lat;

    public:
        dreptunghi(const dreptunghi&);
        int get_lun(void) const;
        int get_lat(void) const;
        void afis() const;
};
```

```
Dreptunghi::Dreptunghi(const Dreptunghi& d)
{
    lun=d.lun;
    lat=d.lat;
}
```

```
int Dreptunghi::get_lun() const
{
    return lun;
}
int Dreptunghi::get_lat() const
{
    return lat;
}
void Dreptunghi::afis() const
{
    cout<<lun<<" "<<lat<<end;
}
```

**//Nu se modifica valoarea atributelor.** Expresii de genul:  
//lun = 0; duc la erori la compilare

Pot sa declar obiecte const.

DAR ! Atentie la utilizare. Trebuie garantat ca nu o sa se incerce modificarea lor.

//utilizare

Dreptunghi p;

const Dreptunghi cv(p);

cv.afis(); //pot sa fac acest apel doar pentru ca functia de afis a fost declarata  
//constanta, altfel nu ar fi avut dreptul sa utilizeze|obiectul constant cv.

## C3: Introducere in programarea obiectuala

Functiile membru de tip const exista deoarece C++ permite crearea unor obiecte de tip const sau a unor referinte catre obiecte de tip const ce vor fi transmise ca parametru. (De asemenea, ele garanteaza, in general, ca functia nu modifica attributele obiectului care o apelaza.)

Pentru obiectele declarate const pot fi apelate doar metode **const** ce nu modifica datele.

Singura exceptie de la aceasta regula este facuta in cazul constructorilor si destructorilor.

Posibilitatea apelarii constructorilor este comparabila cu definitia unei variabile:

```
int const max = 10;
```



## C3: Introducere in programarea obiectuala

### Atribute si functii membru statice:

Fiecare obiect de tipul unei clase are valori proprii ale atributelor.

Totusi, este posibila definirea unor atribute care sa fie folosite in comun de catre toate obiectele unei clase.

Acestea trebuiesc declarate ca **atribute statice**.

Ele exista intr-o singura copie, folosita in comun de toate obiectele instantiate.

Crearea, instantierea si accesul la ele sunt total independente de obiectele clasei.

Un membru static al clasei poate fi indicat folosind numele clasei si operatorul de rezolutie.

**Funcțiile membre statice** efectuează operații care nu sunt asociate obiectelor individuale (membrilor nestatici ai clasei).

Apelarea lor NU se face prin intermediul unui obiect, ci cu numele clasei și operatorul de rezoluție.

Nu pot utiliza `this` în implementarea acestor funcții.

Pot accesa doar date și funcții statice.

Nu pot fi declarate `const` sau `volatile`.

Se pot utiliza înainte de instanțierea unui obiect.

```

class ex_static
{
private:
    static int cateObj; //atribut static - cate obiecte am instantiate la un moment dat
    int x;             //nr. obiectului instantiat
public:
    ex_static() //Constructor
    {   cateObj=cateObj+1;    //incrementez nr de obiecte
        x=cateObj;           //acest obiect are nr -...
    }
    ~ex_static() //Destructor
    {   cateObj=cateObj-1;    //la apelul destructorului obiectul elibereaza spatiul de memorie ocupat
    }
    static void cateAm() //functie statica - manipuleaza date statice
    {   cout << "Nr Obiecte: " << cateObj<<endl; //Pot sa fac afisarea: cout<<x? De ce?
    }
    void number() //functie membra
    {   cout << "Numarul obiectului este: " << x<<endl; //pot sa utilizez date statice: cout<<cateObj;
    }
};

int ex_static::cateObj=0; //initializarea atributului static
int main()
{
    ex_static e1;
    ex_static::cateAm(); //functia statica este accesata folosind numele clasei si operatorul de rezolutie
    ex_static e2,e3,e4;
    ex_static::cateAm();
    e1.number(); e2.number(); //functia membra nestatica e accesata cu numele unui obiect si operatorul de acces (.)
    ex_static *e5=new ex_static;
    ex_static::cateAm(); //5
    delete e5;
    ex_static::cateAm(); //4
    return 0; }

```

## Problema 1

Acolo unde considerati necesar, declarati functiile/argumentele const

```
class vector {  
    // private:  
        int n;                // dimensiune  
        int *vec;             // vectorul de intregi  
    public:  
  
    vector(int n=0);  
    vector(int, int*);  
    vector (vector&);  
    ~vector();  
    vector& copie(vector &);  
    int getdim();  
    void set_elm(int, int);  
    int get_elm(int);  
    void afis();  
    int* getvec();  
};
```

```

class vector {
    // private:
        int n;                // dimensiune
        int *vec;             // vectorul de intregi
    public:

    vector(const int n=0);
    vector(const int, const int*);
    vector (const vector&);
    ~vector();
    vector& copie(const vector &);
    int getdim() const;
    void set_elm(const int, const int);
    int get_elm(const int) const;
    void afis() const;
    int* getvec() const;
};

```

//ce functii as putea sa declar inline?

## **Problema 2**

Ce caracteristici si comportamente pot sa fie evidentiata pentru o colectie de obiecte de tip stiva (LIFO)?

## **Problema 3**

Ce caracteristici si comportamente sunt necesare pentru o colectie de obiecte de tip student (class student) – prin prisma titularului de curs (ex la POO) care doreste sa le tina o evidenta completa a studentilor sai. Implementati o aplicatie care tine aceasta evidenta.

Tema – implementati problema 2 folosindu-va de cursul de SDA