

Capitolul 2. Clase & Obiecte

Concepte de bază

Concepte de bază

def

Clasă = tip de date definit de utilizator care conține date și funcții membre. Reprezintă fundația POO.

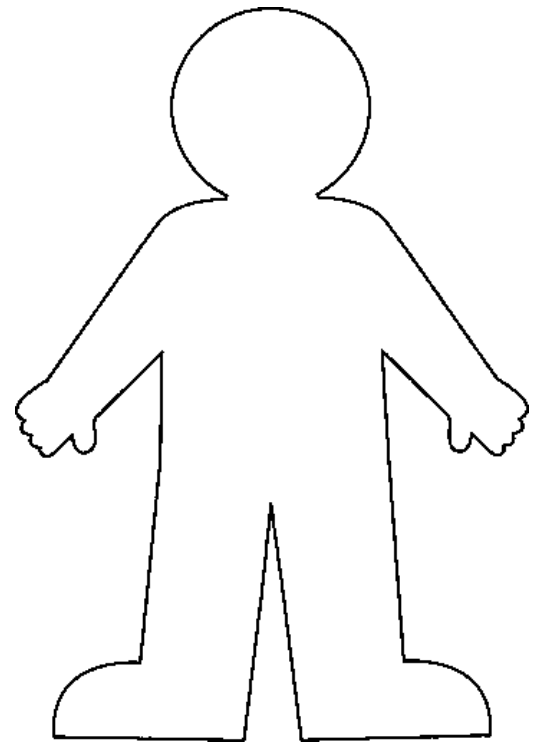
Clasă = tipar/șablon/plan/matriță (blueprint). Nu ocupă memorie.

Date membre (attribute) – reprezintă **starea** clasei

Funcții membre (metode) – reprezintă **comportamentul** clasei

Sintaxă

```
class <nume_clasa> {  
    <specificator_de_acces>:  
        <date_membre>  
        <functii_membre>  
};
```



Principiile POO

1. Încapsulare = gruparea/învelirea/încapsularea datelor și a funcțiilor ce acționează asupra acestora într-un singur container (clasă).
2. Abstractizare
3. Moștenire
4. Polimorfism

Concepte de bază

def

Obiect = instanță a unei clase.

Clasa indică modul general în care vor arăta obiectele. Obiectele vor fi realizări particulare ale clasei.

Sintaxă

`<nume_clasa> <nume_obiect>;` `int var;`

Nume variabilă

Tip de date

The diagram illustrates the mapping of the syntax `<nume_clasa> <nume_obiect>;` to the variable declaration `int var;`. A blue arrow points from the label 'Nume variabilă' (Variable name) to the `<nume_obiect>` placeholder in the syntax and to the `var` variable in the code. Another blue arrow points from the label 'Tip de date' (Data type) to the `<nume_clasa>` placeholder in the syntax and to the `int` data type in the code.

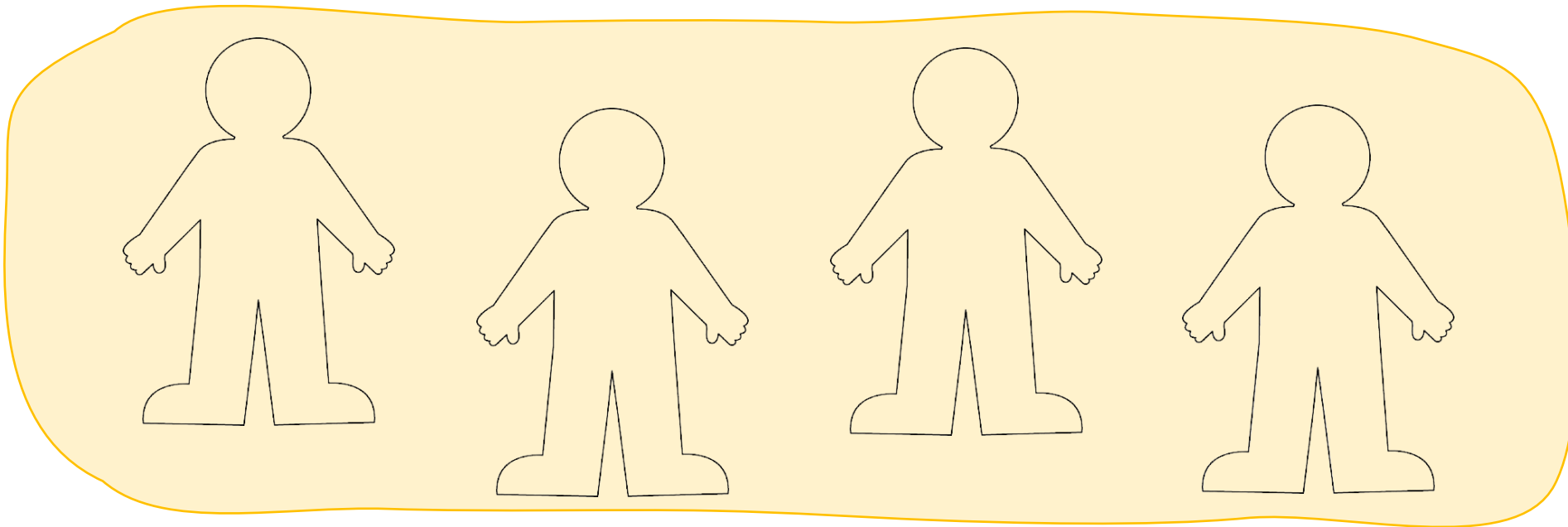
Fiecare obiect primește o copie a tuturor datelor membre.

Obiectele ocupă memorie.

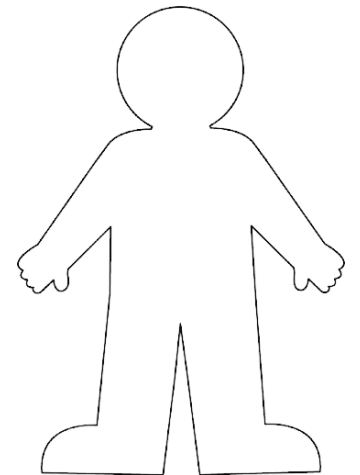


Concepte de bază

Obiecte:

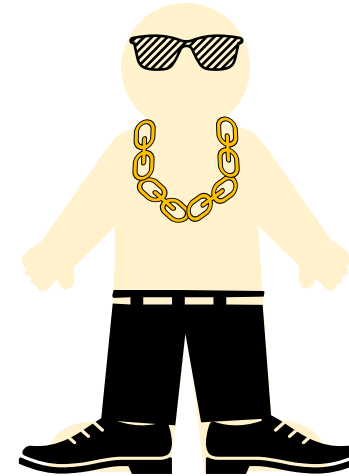
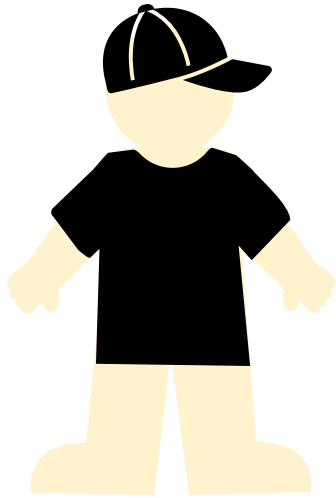


Clasă

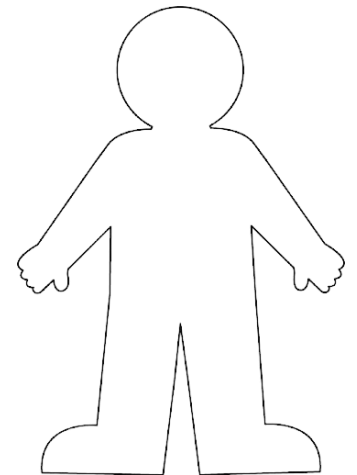
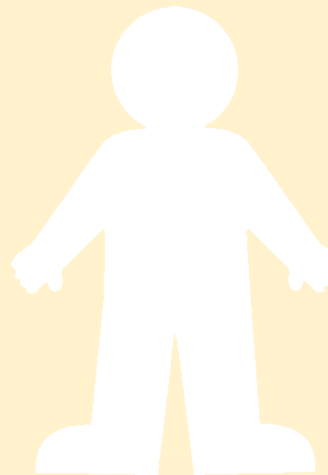
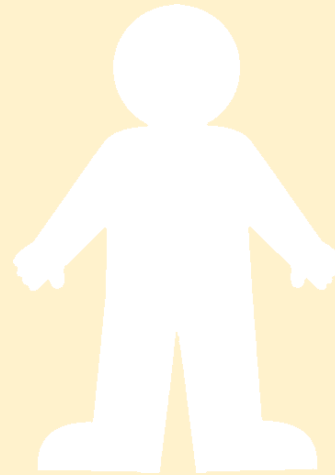
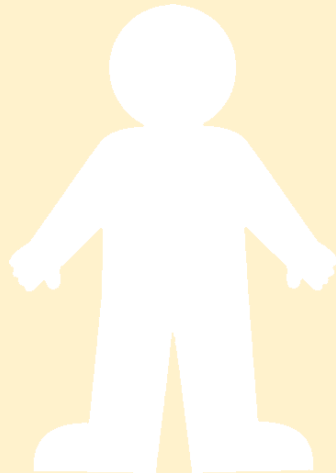
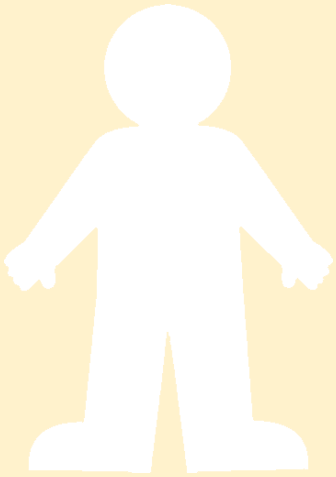


Concepte de bază

Obiecte:



Clasă



Exemplu

Date membre:

1. culoare
2. marcă
3. an
4. nr. km
5. preț



Exemplu

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

```
class Masina{
public:
    std::string culoare, marca;
    int an, km, pret;
};
```

cuvânt cheie

nume clasă

specificator de acces

date membre

```
int main(){
    Masina m;
```

```
    std::cin >> m.culoare;
    std::cin >> m.marca;
    std::cin >> m.an;
    std::cin >> m.km;
    std::cin >> m.pret;
```

Accesul la datele membre (citire/scriere) se face cu operatorul '.'

```
    std::cout << "Culoare:" << m.culoare << std::endl;
    std::cout << "Marca:" << m.marca << std::endl;
    std::cout << "An:" << m.an << std::endl;
    std::cout << "km:" << m.km << std::endl;
    std::cout << "Pret:" << m.pret << std::endl;
```

```
    return 0;
```

```
}
```

Soluția nu este elegantă, deoarece încalcă principiul încapsulării!

Exemplu

Date membre:

1. culoare
2. marcă
3. an
4. nr. km
5. preț

Funcții membre:

1. inițializare date
2. afișare date



Exemplu

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

```
class Masina{
public:
    std::string culoare, marca;
    int an, km, pret;

    void init(std::string culoare, std::string marca,
              int an, int km, int pret){
        culoare = culoare;
        marca = marca;
        an = an;
        km = km;
        pret = pret;
    }
};
```

```
void display_info(){
    std::cout << culoare << std::endl;
    std::cout << marca << std::endl;
    std::cout << an << std::endl;
    std::cout << km << std::endl;
    std::cout << pret << std::endl;
}
```

```
};
```

asignări ambigue
culoare = dată membră sau argument al funcției?

funcții membre

Soluția 1 – schimbare nume argumente

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

class Masina{
public:
    std::string culoare, marca;
    int an, km, pret;

    void init(std::string cul, std::string mk,
              int a_an, int a_km, int a_pret){
        culoare = cul;
        marca = mk;
        an = a_an;
        km = a_km;
        pret = a_pret;
    }

    void display_info(){
        std::cout << culoare << std::endl;
        std::cout << marca << std::endl;
        std::cout << an << std::endl;
        std::cout << km << std::endl;
        std::cout << pret << std::endl;
    }
};
```

Soluția 2 – date membre prefixate

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

class Masina{
public:
    std::string m_culoare, m_marca;
    int m_an, m_km, m_pret;

    void init(std::string culoare, std::string marca,
              int an, int km, int pret){
        m_culoare = culoare;
        m_marca = marca;
        m_an = an;
        m_km = km;
        m_pret = pret;
    }

    void display_info(){
        std::cout << m_culoare << std::endl;
        std::cout << m_marca << std::endl;
        std::cout << m_an << std::endl;
        std::cout << m_km << std::endl;
        std::cout << m_pret << std::endl;
    }
};
```

Soluția 3 – pointer 'this'

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

class Masina{
public:
    std::string culoare, marca;
    int an, km, pret;

    void init(std::string culoare, std::string marca,
              int an, int km, int pret){
        this->culoare = culoare;
        this->marca = marca;
        this->an = an;
        this->km = km;
        this->pret = pret;
    }

    void display_info(){
        std::cout << culoare << std::endl;
        std::cout << marca << std::endl;
        std::cout << an << std::endl;
        std::cout << km << std::endl;
        std::cout << pret << std::endl;
    }
};
```

```
int main () {
    std::string v_culoare = "ALB";
    std::string v_marca = "Dacia";
    int v_an = 2021;
    int v_km = 32000;
    int v_pret = 25000;

    Masina m;
    m.init(v_culoare, v_marca, v_an, v_km, v_pret);
    m.an = 2020;
    m.display_info();

    return 0;
}
```



this = pointer implicit către obiectul a cărui funcție se execută.

this este argument transmis implicit către toate funcțiile membre non-statice.

```
ALB
Dacia
2020
32000
25000

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

Specificatori de acces

1. public
2. protected
3. private

Specificatori de acces

def

Specificatori de acces = cuvinte cheie care modifică drepturile de a accesa membrii (datele sau funcțiile) unei clase:

1. `public`: membrii clasei pot fi accesați de oriunde este vizibilă clasa.
2. `protected`: membrii clasei pot fi accesați de membrii aceleiași clase (+ 'prietenii') și de către membrii claselor derivate. (@moștenire)
3. `private`: membrii clasei pot fi accesați doar de membrii aceleiași clase (+ 'prietenii')

Default: `private`

Efectul unui specificator de acces durează până la întâlnirea unui nou specificator de acces sau până la finalul clasei (oricare eveniment apare primul).

Specificatori de acces

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

class Masina{
private:
    std::string culoare, marca;
    int an, km, pret;

public:
    void init(std::string culoare, std::string marca,
              int an, int km, int pret){
        this->culoare = culoare;
        this->marca = marca;
        this->an = an;
        this->km = km;
        this->pret = pret;
    }

    void display_info(){
        std::cout << culoare << std::endl;
        std::cout << marca << std::endl;
        std::cout << an << std::endl;
        std::cout << km << std::endl;
        std::cout << pret << std::endl;
    }
};
```

```
int main () {
    std::string v_culoare = "ALB";
    std::string v_marca = "Dacia";
    int v_an = 2021;
    int v_km = 32000;
    int v_pret = 25000;

    Masina m;

    m.init(v_culoare, v_marca, v_an, v_km, v_pret);
    m.an = 2020;
    m.display_info();

    return 0;
}
```

In function 'int main()':

error: 'int Masina::an' is private within this context

note: declared private here

=== Build failed: 1 error(s), 0 warning(s) (0 minute(s).

“Soluție”



Folosiți `public` în specificatorul de acces public pentru toate componentele clasei.



“Soluție”

Scenariu 1: aplicație bancară

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

```
class Cont{
public:
    std::string nume;
    float sold;

    void initializare(std::string nume, float sold){
        this->nume = nume;
        this->sold = sold;
    }
};
```

```
int main () {
    Cont un_cont;
    un_cont.initializare("Mihai", 1000);
```

Cod scris în aplicația bancară

```
un_cont.sold += 1000;
```

Cod scris de client

```
std::cout << un_cont.sold; // afiseaza 2000
return 0;
}
```



“Soluție”

Scenariu 2: bază de date studenți

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


```
class Student{
public:
    std::string nume;
    long long cnp;
    char seria;
    short anul;
    std::string email;
    ...
};
```

```
int main () {
```

```
    ...
    for (int i=0; i<1000; ++i) {
        std::cout << student[i].nume << ' '
                    << student[i].cnp << ' '
                    << student[i].seria << ' '
                    << student[i].anul << ' '
                    << student[i].email << '\n';
    }
```

```
    return 0;
```

Oricine poate avea acces la datele studenților

 Data hiding = procesul de a “ascunde” (restricționa accesul la) datele membre față de restul lumii.

Pe cât posibil, facem toate datele membre de tip `private`.

Pentru a le accesa/modifica, creăm funcții membre (getters/setters) de tip `public` care interacționează cu acestea.

Exemplu

```
class Rectangle{
private:
    float width;
    float height;

public:
    void set_width(float width){
        this->width = width;
    }

    float get_width(){
        return this->width;
    }

    void set_height(float height){
        this->height = height;
    }

    float get_height(){
        return this->height;
    }
};
```

- Datele membre <data> sunt modificate prin funcțiile set_<data>
- Datele membre <data> sunt accesate prin funcțiile get_<data>
- Datele membre sunt `private`
- Funcțiile care manipulează datele membre sunt `public`

Exemplu

```
class Rectangle{
private:
    float width;
    float height;

public:
    void set_width(float width){
        this->width = width;
    }

    float get_width(){
        return this->width;
    }

    void set_height(float height){
        this->height = height;
    }

    float get_height(){
        return this->height;
    }
};
```

```
#include<iostream>

class Rectangle {...};

int main(){
    Rectangle r;

    r.set_width(5);
    r.set_height(10);

    std::cout << "width=" << r.get_width() << '\n';
    std::cout << "height=" << r.get_height() << '\n';

    return 0;
}
```

Nu mai există funcție de inițializare?

Fiecare dată membră trebuie inițializată independent?

Constructori

1. implicit
2. parametrizat
3. de conversie
4. de delegare
5. de copiere
6. de mutare

Constructor (ctor)



Constructor = funcție membră specială a unei clase, cu același nume cu clasa, folosită pentru a inițializa datele membre ale unui obiect.

- Este executat automat atunci când se creează un obiect al clasei respective.
- Nu se poate apela explicit dintr-o instanță a clasei.
- Nu are tip de date returnat (nici măcar `void`).

1. Constructor implicit (default ctor)

```
#include <iostream>

class Rectangle{
private:
    float width;
    float height;

public:
    // Constructor default fara parametri:
    Rectangle(){
        this->width = 0;
        this->height = 0;
    }

    // getters & setters:
    ...
};

int main(){
    Rectangle r; // apel constructor default

    std::cout << "width:" << r.get_width() << "\n"; // afiseaza 0
    std::cout << "height:" << r.get_height() << "\n"; // afiseaza 0

    return 0;
}
```

2. Constructor parametrizat (parameterised ctor)

```
#include <iostream>
```

```
class Rectangle{  
private:  
    float width;  
    float height;  
  
public:  
    // Constructor parametrizat cu doi parametri:  
    Rectangle(float width, float height){  
        this->width = width;  
        this->height = height;  
    }  
  
    // getters & setters:  
    ...  
};
```

Existența unui constructor parametrizat suprimă constructorul default

```
int main(){  
    Rectangle r1; // eroare: constructorul default nu mai exista  
    Rectangle r2(7, 10); // width = 7, height = 10  
  
    return 0;  
}
```

2. Constructor parametrizat (parameterised ctor)

```
#include <iostream>

class Rectangle{
private:
    float width;
    float height;

public:
    // Constructor default cu toți parametrii având valori default:
    Rectangle(float width=0, float height=0){
        this->width = width;
        this->height = height;
    }

    // getters & setters:
    ...
};

int main(){
    Rectangle r1; // width = 0, height = 0
    Rectangle r2(5); // width = 5, height = 0
    Rectangle r3(7, 10); // width = 7, height = 10

    return 0;
}
```

Funcție cu valori implicite ⇔ la momentul apelului, dacă nu este transmisă o valoare pentru argumentele respective, atunci acestea sunt inițializate cu valorile implicite din lista de argumente.

În timpul apelului funcției, argumentele sunt copiate de la stânga la dreapta.

3. Constructor de conversie (conversion ctor)

```
#include <iostream>

class Rectangle{
private:
    float width;
    float height;

public:
    // Constructor cu un singur parametru
    Rectangle(float width){
        this->width = width;
        this->height = width;
    }

    // getters & setters:
    ...
};

int main(){
    Rectangle r1 = 3.0f; // width = 3.0, height = 3.0

    return 0;
}
```

Dacă un constructor nu este declarat explicit și are un singur parametru, atunci tipul parametrului poate fi convertit implicit la tipul clasei.

3. Constructor de conversie (conversion ctor)

```
#include <iostream>

class Rectangle{
private:
    float width;
    float height;

public:
    // Constructor cu un singur parametru neimplicit
    Rectangle(float width, float height=0){
        this->width = width;
        this->height = height;
    }

    // getters & setters:
    ...
};

int main(){
    Rectangle r1 = 3.0f; // width = 3.0, height = 0

    return 0;
}
```

Dacă un constructor nu este declarat `explicit` și toți parametrii săi sunt declarați cu valori implicite, mai puțin unul singur, atunci tipul parametrului poate fi convertit implicit la tipul clasei.

3. Constructor de conversie (conversion ctor)

```
#include <iostream>

class Rectangle{
private:
    float width;
    float height;

public:
    // Constructor cu un singur parametru neimplicit
    explicit Rectangle(float width, float height=0){
        this->width = width;
        this->height = height;
    }

    // getters & setters:
    ...
};

int main(){
    Rectangle r1 = 3.0f; // eroare de compilare

    return 0;
}
```

Cuvântul cheie `explicit` previne conversiile implicite și anulează constructorul de conversie.

Funcții supraîncărcate



Funcții supraîncărcate = două (sau mai multe) funcții care au același nume, dar diferă prin lista de parametri (fie prin tipul lor, fie prin numărul lor).

Funcții supraîncărcate - exemplu

```
#include <iostream>

void print_data(int i){
    std::cout << "Afisare int:" << i << std::endl;
}

void print_data(float f){
    std::cout << "Afisare float:" << f << std::endl;
}

void print_data(int i, float f){
    std::cout << "Afisare int si float:" << i << " " << f << std::endl;
}

int main () {
    print_data(3);
    print_data(3.5f);
    print_data(3, 3.3f);

    return 0;
}
```

```
Afisare int:3
Afisare float:3.5
Afisare int si float:3 3.3
```

```
Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

Constructorii supraîncărcați (overloaded ctor)

```
class Rectangle{
private:
    float width;
    float height;

public:
    Rectangle(){
        std::cout << "Constructor default.\n";
        this->height = 0;
        this->width = 0;
    }
    Rectangle(float x){
        std::cout << "Constructor cu param float.\n";
        this->width = x;
        this->height = x;
    }
    Rectangle(int x){
        std::cout << "Constructor cu param int.\n";
        this->width = sqrt(x);
        this->height = sqrt(x);
    }
    Rectangle(float width, float height){
        std::cout << "Constructor cu 2 parametri.\n";
        this->width = width;
        this->height = height;
    }
};
```

```
#include <iostream>
#include <cmath>
```

```
class Rectangle{...}
```

```
int main () {
    Rectangle r1; // width = 0, height = 0
    Rectangle r2(5.23f); // width = 5.23, height = 5.23
    Rectangle r3(25); // width = 5, height = 5
    Rectangle r4(3, 10); // width = 3, height = 10
    return 0;
}
```

```
Constructor default.
Constructor cu param float.
Constructor cu param int.
Constructor cu 2 parametri.
```

```
Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

4. Constructori de delegare (delegating ctor) - C++11

```
#include <iostream>
```

```
class Box{  
private:  
    int length, width, height;  
public:  
    Box(int length){  
        this->length = (length > 0) ? length : 0;  
    }  
    Box (int length, int width){  
        this->length = (length > 0) ? length : 0;  
        this->width = (width > 0) ? width : 0;  
    }  
    Box (int length, int width, int height){  
        this->length = (length > 0) ? length : 0;  
        this->width = (width > 0) ? width : 0;  
        this->height = (height > 0) ? height : 0;  
    }  
};
```

```
int main(){  
    Box b1(5); // length = 5  
    Box b2(10, -3); // length = 10, width = 0  
    Box b3(-2, 3, 4); // length = 0, width = 3, height = 4  
    return 0;  
}
```

Blocuri de cod care se repetă => redundanță inutilă

4. Constructori de delegare (delegating ctor) - C++11

```
#include <iostream>
```

```
class Box{
private:
    int length, width, height;
public:
    Box(int length){
        this->length = (length > 0) ? length : 0;
    }
    Box (int length, int width):Box(length){
        this->width = (width > 0) ? width : 0;
    }
    Box (int length, int width, int height):Box(length, width){
        this->height = (height > 0) ? height : 0;
    }
};

int main(){
    Box b1(6); // length = 5
    Box b2(10, -3); // length = 10, width = 0
    Box b3(-1, 3, 4); // length = 0, width = 3, height = 4

    return 0;
}
```

Constructorii deleagă o parte din sarcini către alți constructori (cu mai puțini parametri)

4. Constructori de delegare (delegating ctor) - C++11

```
#include <iostream>
```

```
class Box{  
private:  
    int length, width, height;  
public:  
    Box(int length){  
        this->length = (length > 0) ? length : 0;  
    }  
    Box (int length, int width):Box(length, width, 3){  
        this->width = (width > 0) ? width : 0;  
    }  
    Box (int length, int width, int height):Box(length, width){  
        this->height = (height > 0) ? height : 0;  
    }  
};
```

Atenție la eventuale bucle!

5. Constructori de copiere (copy ctor)

def

Constructorii de copiere = funcții membre care inițializează obiecte folosind datele unui alt obiect din aceeași clasă, inițializat anterior.

```
class Rectangle{
private:
    float width;
    float height;

public:
    Rectangle(float width, float height){
        std::cout << "Constructor cu 2 parametri.\n";
        this->width = width;
        this->height = height;
    }

    Rectangle(const Rectangle &r){
        std::cout << "Constructor de copiere.\n";
        this->width = r.width;
        this->height = r.height;
    }
};
```

```
#include <iostream>

class Rectangle{...};

int main(){
    Rectangle r1(5, 3);
    Rectangle r2(r1); //r2.width=5; r2.height=3

    return 0;
}
```

```
Constructor cu 2 parametri.
Constructor de copiere.
```

```
Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```

5. Constructori de copiere (copy ctor)

```
Rectangle(const Rectangle &r) {  
    std::cout << "Constructor de copiere.\n";  
    this->width = r.width;  
    this->height = r.height;  
}
```

- cuvântul cheie `const` – vrem să ne asigurăm că atunci când facem copierea nu modificăm obiectul pe care îl copiem (referința); prin urmare îl obligăm să rămână constant în domeniul constructorului.
- referința (`Rectangle &r`) – un nume alternativ pentru o variabilă existentă. Se obține prin prefixarea identificatorului (numelui) variabilei cu simbolul `&`.
 - nu forțează crearea unei copii suplimentare în funcția apelantă, ci folosește o indirectare către un obiect deja existent.
 - nu au nevoie de operator de dereferențiere pentru accesarea valorii.
 - membrii unei referințe pot fi accesați cu operatorul `.'`, fără să fie nevoie de `'->'`.

Ce s-ar întâmpla dacă nu s-ar folosi referință?

5. Constructori de copiere (copy ctor)

```
class Student{
private:
    int *note;
    int nr_note;
public:
    Student(int *note, int nr_note){
        this->nr_note = nr_note;
        this->note = new int[nr_note];
        for (int i=0; i<nr_note; ++i){
            *(this->note+i) = *(note+i);
        }
    }
    Student(Student &s){
        this->nr_note = s.nr_note;
        this->note = s.note;
    }
    void display(){
        for (int i=0; i<nr_note; ++i){
            std::cout << *(note+i) << " ";
        }
    }
    void increment(int incr){
        for(int i=0; i<nr_note; ++i){*(note+i) += incr;}
    }
};
```

Operator care alocă memorie
și returnează pointer către
începutul blocului alocat.

Copierea implicită se face
element cu element

```
#include <iostream>

class Student{...};

int main(){
    int nr_note;
    std::cin >> nr_note;
    int *note = new int[nr_note];
    for(int i=0; i<nr_note; i++){
        std::cin >> *(note+i);
    }
    Student s1(note, nr_note);
    Student s2(s1);
    s1.display();
    std::cout<<std::endl;
    s2.increment(5);
    s1.display();

    return 0;
}
```

```
3
1 5 7
1 5 7
6 10 12
Process returned 0 (0x0)   execution time : 4.512 s
Press any key to continue.
```


5. Constructori de copiere (copy ctor)

```
class Student{
private:
    int *note;
    int nr_note;
public:
    Student(int *note, int nr_note){
        this->nr_note = nr_note;
        this->note = new int[nr_note];
        for (int i=0; i<nr_note; ++i){
            *(this->note+i) = *(note+i);
        }
    }
    Student(Student &s){
        this->nr_note = s.nr_note;
        this->note = new int[this->nr_note];
        for (int i=0; i<nr_note; ++i){
            *(this->note+i) = *(s.note+i);
        }
    }
    void display(){
        for (int i=0; i<nr_note; ++i){
            std::cout << *(note+i) << " ";
        }
    }
    void increment(int incr){
        for(int i=0; i<nr_note; ++i){*(note+i) += incr;}
    }
};
```

```
#include <iostream>

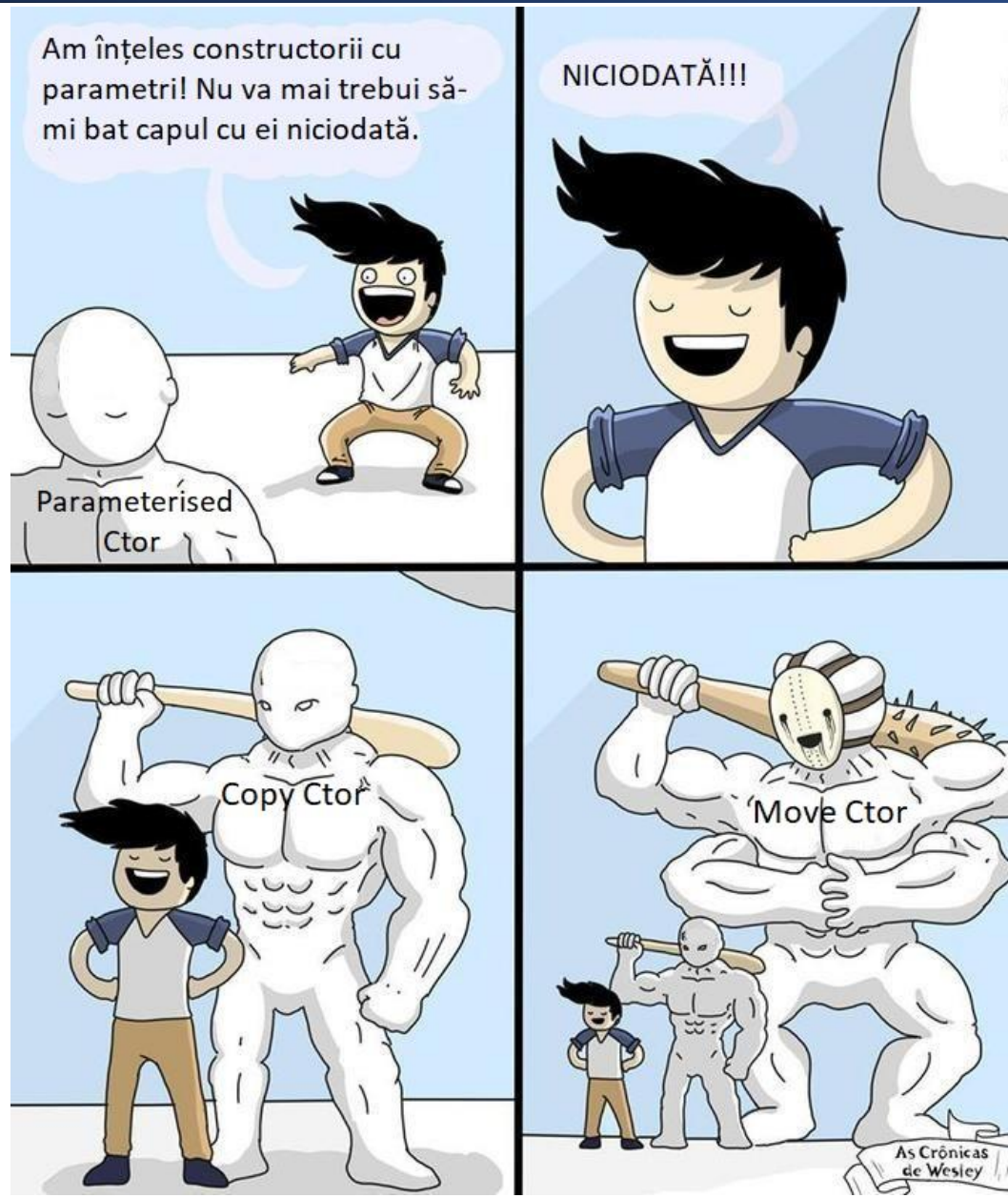
class Student{...};

int main(){
    int nr_note;
    std::cin >> nr_note;
    int *note = new int[nr_note];
    for(int i=0; i<nr_note; i++){
        std::cin >> *(note+i);
    }
    Student s1(note, nr_note);
    Student s2(s1);
    s1.display();
    std::cout<<std::endl;
    s2.increment(5);
    s1.display();

    return 0;
}
```

```
3
1 5 7
1 5 7
1 5 7
Process returned 0 (0x0)   execution time : 4.559 s
Press any key to continue.
```

6. Construtores de mutare (move ctor) – C++11



6. Constructori de mutare (move ctor)

def

- `lvalue` = locator value; o valoare care ocupă o locație identificabilă (are o adresă) în memorie. Se pot face asignări către `lvalues`.

def

- `rvalue` = tot ce nu este `lvalue`. O expresie este `rvalue` dacă rezultă într-un obiect temporar. Nu se pot face asignări către `rvalues`.

```
int a, var;  
var = 4;      // var=lvalue; 4=rvalue  
4 = var;      // 4!=lvalue;  var=lvalue  
var + 1 = 4;  // var=lvalue, var+1!=lvalue  
var = 4 - 1;  // var=lvalue; 4-1=rvalue  
a = var + 1;  // a=lvalue;    var=lvalue;    var+1=rvalue  
a - var = 1;  // a=lvalue;    var=lvalue;    a-var=rvalue
```

6. Constructori de mutare (move ctor)

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

```
class Student{
private:
    std::string *nume;
public:
    Student(const std::string& str){
        this->nume = new std::string(str);
    }
    Student(Student &&s){
        this->nume = s.nume;
        s.nume = nullptr;
    }
    Student(Student &s){
        this->nume = new std::string(*s.nume);
    }
    void display(){std::cout << *nume << std::endl;}
};

Student f(Student s){return s;}
int main(){
    Student s1("John");
    Student s2 = f(s1);
    s1.display();
    s2.display();
    return 0;
}
```

Alocare bloc memorie, copiere `str` în zona alocată și returnare pointer către blocul de memorie.

Referință la o `rvalue`.

Accesul la membri se realizează cu operatorul `.`

Asignarea `nullptr` pentru pointer-ul `*nume` din referința la `rvalue` previne ștergerea accidentală a datelor din memorie atunci când obiectul din `rvalue` este distrus.

Funcție pentru a transforma un obiect în `rvalue`.

Apel constructor de mutare.

Liste de inițializare

Liste de inițializare

Liste de inițializare = un mod de a inițializa datele membre în definiția constructorilor.

Utilizare:

- Inițializarea datelor membre constante non-stactice
- Inițializarea datelor membre de tip referință
- Inițializarea obiectelor membre care nu au constructor default
- Inițializarea datelor membre cu parametri ce au același nume (alternativă la pointer-ul `this`)


Liste de inițializare

```
#include <iostream>
class Example{
private:
    int m;
    int &r;
    const int c;
public:
    Example(int m, int &referinta, const int constanta){
        this->m = m;
        this->r = referinta; // eroare de compilare
        this->c = constanta; // eroare de compilare
    }

    void display(){
        std::cout << "m:" << m << std::endl;
        std::cout << "r:" << r << std::endl;
        std::cout << "c:" << c << std::endl;
    }
};

int main(){
    int a = 20;
    const int b = 30;
    Example e(10, a, b);
    e.display();
    return 0;
}
```

Obiectul se creează până să se execute blocul de cod aferent constructorului. Constructorul doar inițializează datele.



Liste de inițializare

```
#include <iostream>
class Example{
private:
    int m;
    int &r;
    const int c;
public:
    Example(int m, int &referinta, const int constanta):m(m), r(referinta), c(constanta){}

    void display(){
        std::cout << "m:" << m << std::endl;
        std::cout << "r:" << r << std::endl;
        std::cout << "c:" << c << std::endl;
    }
};

int main(){
    int a = 20;
    const int b = 30;
    Example e(10, a, b);
    e.display();
    return 0;
}
```

```
m:10
r:20
c:30
```

```
Process returned 0 (0x0)   execution time : 0.024 s
Press any key to continue.
```


Destructori

Destructorii



“For every action there is an equal and opposite reaction.”
Sir Isaac Newton

Destructor

def

Destructor = funcție membră specială a unei clase, cu același nume cu clasa, însă precedată de `~`, folosită pentru a elibera resursele ce au fost ocupate de obiect în timpul existenței sale.

Destructorul este apelat implicit la finalul duratei de viață a unui obiect. De obicei, nu se apelează explicit.

Reguli generale:

- pentru fiecare `new` avem nevoie de un `delete`;
- pentru fiecare `new []` avem nevoie de un `delete []`;
- pentru fiecare `malloc()` avem nevoie de `free()`.

Destructorii

```
class Student{
private:
    int *note;
    int nr_note;
public:
    Student(int *note, int nr_note){
        this->note = new int[nr_note];
        for (int i=0; i<nr_note; i++){
            *(this->note+i) = *(note+i);
            this->nr_note = nr_note;
        }
    }

    ~Student(){
        delete [] note;
        std::cout << "Apel destructor!" << std::endl;
    }

    void display(){
        for (int i=0; i<this->nr_note; ++i){
            std::cout << *(note+i) << " ";
        }
        std::cout << std::endl;
    }
};
```

```
#include <iostream>
```

```
class Student{...};
```

```
int main(){
    int a[4] = {3, 7, 1, 5};
    Student s1(a, 4);
    s1.display();

    return 0;
}
```

```
3 7 1 5
Apel destructor!
```

```
Process returned 0 (0x0)   execution time : 0.082 s
Press any key to continue.
```

Alocare dinamică a unui vector de întregi
Eliberarea memoriei alocate dinamic pentru un vector

Principiile POO

1. Încapsulare = gruparea/învelirea/încapsularea datelor și a funcțiilor ce acționează asupra acestora într-un singur container (clasă).
2. Abstractizare = procedeul prin care se expun lumii exterioare doar funcționalitățile importante, fără a se intra în prea multe detalii.
3. Moștenire
4. Polimorfism

Vectori de obiecte

1. Vector static de obiecte fără inițializare explicită

```
#include <iostream>

class Point{
    int x, y;
public:
    Point(int x=0, int y=0):x(x), y(y){}
    void display(){
        std::cout << "x=" << x << std::endl;
        std::cout << "y=" << y << "\n\n";
    }
};

int main(){
    Point point_array_1[3]; ←
    for (int i=0; i<3; ++i){
        point_array_1[i].display();
    }

    return 0;
}
```

```
x=0
y=0
```

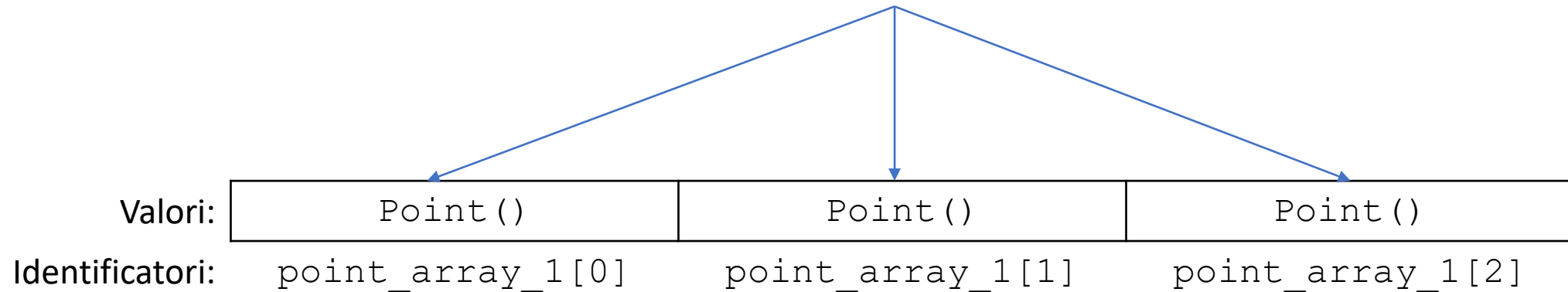
```
x=0
y=0
```

```
x=0
y=0
```

```
Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

1. Vector static de obiecte fără inițializare explicită

Obiecte create folosind constructorul implicit



2. Vector static de obiecte cu inițializare explicită

```
#include <iostream>

class Point{
    int x, y;
public:
    Point(int x=0, int y=0):x(x), y(y){}
    void display(){
        std::cout << "x=" << x << std::endl;
        std::cout << "y=" << y << "\n\n";
    }
};

int main(){
    Point point_array_2[3] = {Point(3, 2),
                              Point(5),
                              Point()};

    for (int i=0; i<3; ++i){
        point_array_2[i].display();
    }

    return 0;
}
```

```
x=3
y=2
```

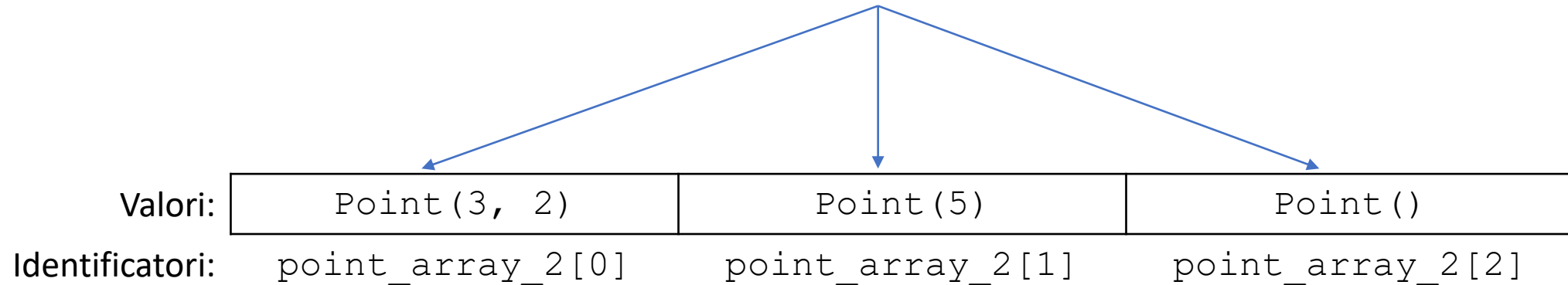
```
x=5
y=0
```

```
x=0
y=0
```

```
Process returned 0 (0x0)   execution time : 0.007 s
Press any key to continue.
```

2. Vector static de obiecte cu inițializare explicită

Obiecte create folosind constructori cu inițializare explicită



3. Vector static de pointeri cu inițializare explicită

```
#include <iostream>

class Point{
    int x, y;
public:
    Point(int x=0, int y=0):x(x), y(y){}
    void display(){
        std::cout << "x=" << x << std::endl;
        std::cout << "y=" << y << "\n\n";
    }
};

int main(){
    Point *point_array_3[3];

    for (int i=0; i<3; ++i){
        point_array_3[i] = new Point(i, i+1);
    }

    for (int i=0; i<3; ++i){
        point_array_3[i] -> display();
        delete point_array_3[i];
    }

    return 0;
}
```

```
x=0
y=1

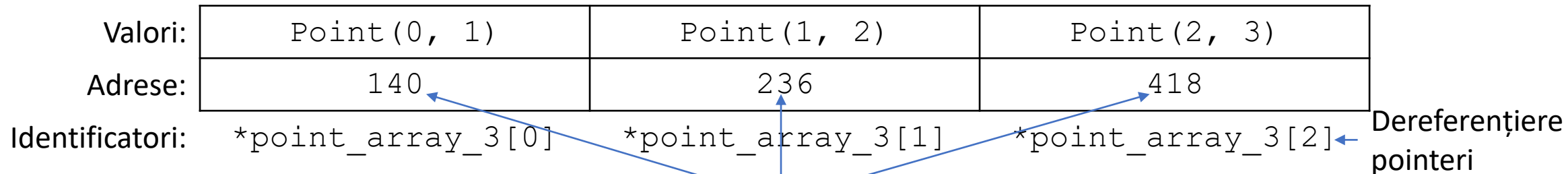
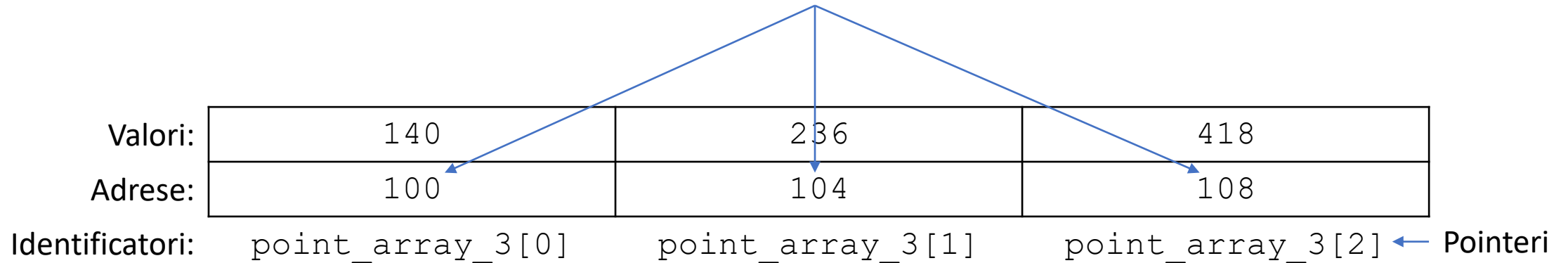
x=1
y=2

x=2
y=3

Process returned 0 (0x0)   execution time : 0.006 s
Press any key to continue.
```

3. Vector static de pointeri cu inițializare explicită

Pointerii sunt creați în locații consecutive de memorie



Obiectele pot fi create în locații aleatoare de memorie

4. Vector dinamic de pointeri cu inițializare explicită

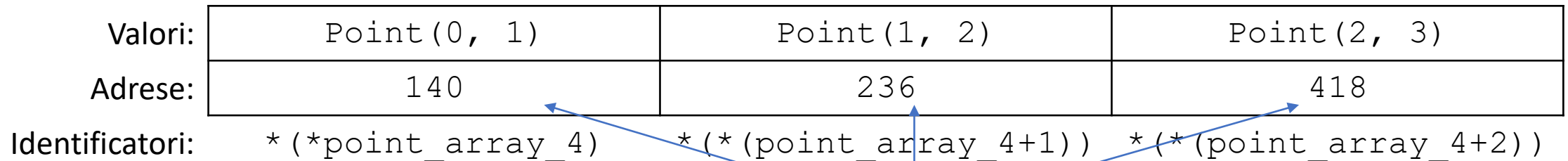
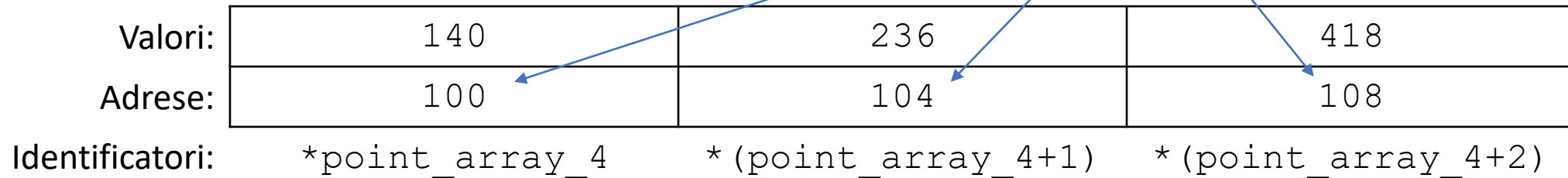
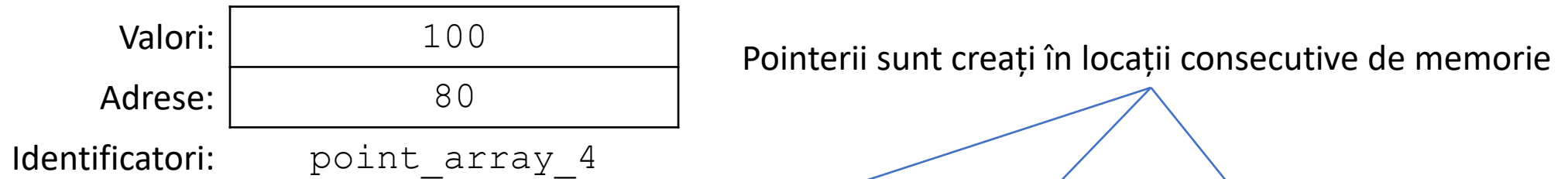
```
#include <iostream>
class Point{
    int x, y;
public:
    Point(int x=0, int y=0):x(x), y(y){}
    void display(){
        std::cout << "x=" << x << std::endl;
        std::cout << "y=" << y << "\n\n";
    }
};
int main(){
    int nr_points;
    Point **point_array_4 = nullptr;
    std::cin >> nr_points;
    point_array_4 = new Point*[nr_points];

    for(int i=0; i<nr_points; ++i){
        *(point_array_4 + i) = new Point(i, i+1);
    }
    for(int i=0; i<nr_points; ++i){
        (*(point_array_4 + i)) -> display();
        delete *(point_array_4 + i);
    }
    delete point_array_4;
    return 0;
}
```

```
3
x=0
y=1
x=1
y=2
x=2
y=3
```

```
Process returned 0 (0x0)   execution time : 1.615 s
Press any key to continue.
```

4. Vector dinamic de pointeri cu inițializare explicită



Obiectele pot fi create în locații aleatoare de memorie

Funcții membre speciale

Funcții membre speciale

```
class Empty{};
```



```
class Empty{  
public:
```

```
    Empty(); // Constructor default  
    Empty(const Empty&); // Constructor de copiere  
    Empty(Empty &&); // Constructor de mutare  
    ~Empty(); // Destructor  
    Empty& operator=(const Empty&); // Operatorul de asignare prin copiere  
    Empty& operator=(Empty&&); // Operatorul de asignare prin mutare  
};
```


Funcții membre speciale

Câteva reguli:

1. Dacă orice constructor este declarat explicit, atunci nu se generează constructorul `default`;
2. Dacă se declară explicit un constructor sau operator de mutare, atunci nu se generează constructorul și nici operatorul de copiere `default`;
3. Dacă se declară explicit un constructor sau operator de copiere, un constructor sau operator de mutare sau destructor, atunci nu se generează constructorul și nici operatorul de mutare `default`.
4. Pentru a forța generarea funcțiilor membre speciale `default` se folosește cuvântul cheie `default`.
5. Pentru a forța suprimarea funcțiilor membre speciale `default` se folosește cuvântul cheie `delete`.

Funcții membre speciale

```
#include <iostream>

class Example{
private:
    int m;
public:
    // Constructorul cu un parametru suprima constructorul default
    Example(int a):m(a){}

    // Generarea explicita a constructorului default
    Example() = default;

    // Suprimare explicita a constructorului de copiere
    // Va genera eroare de compilare daca va fi apelat
    // Anunta programatorul de intentia explicita de a nu folosi acest constructor
    Example(const Example &) = delete;
};
```

Separarea claselor în fișiere

Definire funcții membre în interiorul clasei

```
#include <iostream>

class Minge{
private:
    int raza;
    std::string culoare;
public:
    Minge(int r=0, std::string c=""):raza(r), culoare(c){}
    ~Minge(){std::cout << "Apel destructor" << std::endl;}
    void display_info(){
        std::cout << "Raza:" << raza << std::endl;
        std::cout << "Culoare:" << culoare << std::endl;
    }
};

int main(){
    Minge m(2, "rosu");
    m.display_info();
    return 0;
}
```

Definire funcții membre în afara clasei

```
#include <iostream>
```

```
class Minge{  
private:  
    int raza;  
    std::string culoare;  
public:  
    Minge(int, std::string);  
    ~Minge();  
    void display_info();  
};
```

Minge.h

← Declarare funcții în interiorul clasei

```
Minge::Minge(int r=0, std::string c=""):raza(r), culoare(c){}
```

← Definire funcții în exteriorul clasei

```
Minge::~~Minge(){std::cout << "Apel destructor" << std::endl;}
```

```
void Minge::display_info(){  
    std::cout << "Raza:" << raza << std::endl;  
    std::cout << "Culoare:" << culoare << std::endl;  
}
```

```
int main(){  
    Minge m(2, "rosu");  
    m.display_info();  
    return 0;  
}
```

29/05/22

Programare Obiect-Orientată – Mihai DOGARIU

69

Separarea claselor în fișiere – declarare în header

```
// Minge.h
#ifndef MINGE_H
#define MINGE_H

#include <string>

class Minge{
private:
    int raza;
    std::string culoare;
public:
    Minge(int, std::string);
    ~Minge();
    void display_info();
};

#endif // MINGE_H
```

Include guards

Prototipuri funcții membre

Separarea claselor în fișiere – definire în fișiere sursă

```
// Minge.cpp  
#include "Minge.h"  
#include <iostream>
```

← Includem fișier header clasă

```
Minge::Minge(int r=0, std::string c=""):raza(r), culoare(c){}
```

```
Minge::~Minge() {std::cout << "Apel destructor" << std::endl;}
```

```
void Minge::display_info() {  
    std::cout << "Raza:" << raza << std::endl;  
    std::cout << "Culoare:" << culoare << std::endl;  
}
```

← Definiții funcții membre

Separarea claselor în fișiere – programul principal

```
// main.cpp
```

```
#include "Minge.h"
```

← Includem fișier header clasă

```
int main(){  
    Minge m(2, "rosu");  
    m.display_info();  
    return 0;  
}
```

```
Raza:2
```

```
Culoare:rosu
```

```
Apel destructor
```

```
Process returned 0 (0x0)   execution time : 0.008 s
```

```
Press any key to continue.
```


Cuvântul cheie 'static'

Cuvântul cheie 'static'

1. **Funcțiile sau variabilele** din domeniul fișierului (în namespace sau globale – în afara funcției `main`) declarate `static` au “legare internă” (internal linkage) = sunt accesibile în unitatea de traducere (translation unit = fișierul obținut după procesarea tuturor directivelor `#include`) curentă. În lipsa cuvântului cheie `static`, aceste funcții sau variabile au “legare externă”.

În lipsa unei inițializări explicite, variabilele statice se inițializează cu valoarea 0.

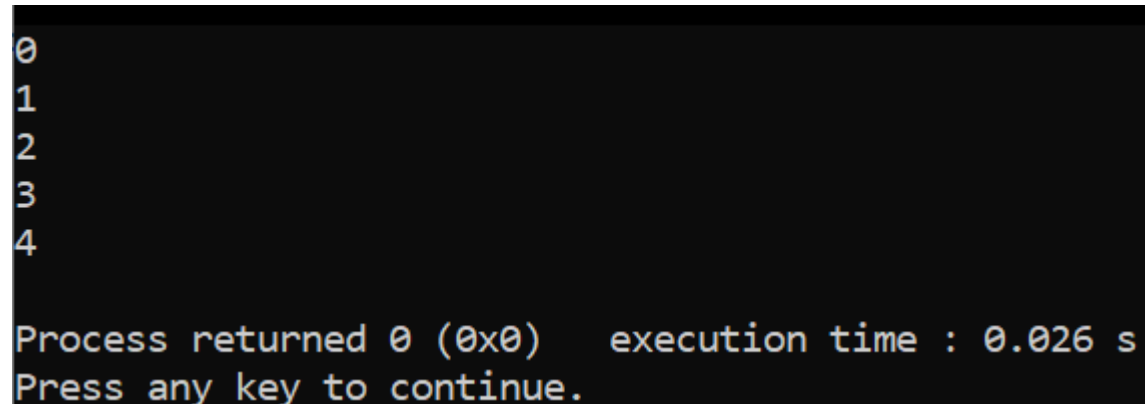
Cuvântul cheie 'static'

2. O **variabilă** declarată `static` **într-o funcție** își menține starea între apelurile succesive ale funcției.

```
#include<iostream>

void f(){
    static int i;
    std::cout << i++ << std::endl;
}

int main(){
    for (int i=0; i<5; ++i){
        f();
    }
    return 0;
}
```



A screenshot of a terminal window showing the output of a C++ program. The output consists of five lines, each containing a number from 0 to 4, representing the value of a static variable 'i' that is incremented in each call to function 'f'. Below the output, the terminal shows the message 'Process returned 0 (0x0) execution time : 0.026 s' and 'Press any key to continue.', indicating the program has finished execution.

```
0
1
2
3
4

Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

Cuvântul cheie 'static'

3. În cadrul unei clase, **o dată membră** declarată `static` înseamnă că toate instanțele clasei vor partaja o singură copie a respectivei date. O dată membră statică trebuie definită în domeniul fișierului.

Cuvântul cheie 'static'

```
#include<iostream>

class Example{
public:
    static int i; // declarare in clasa
};

// definire în domeniul fisierului:
int Example::i;

int main(){
    Example e1, e2;
    std::cout << e1.i << std::endl;
    std::cout << e2.i << std::endl;
    e1.i++;
    std::cout << e1.i << std::endl;
    std::cout << e2.i << std::endl;
    e2.i++;
    std::cout << e1.i << std::endl;
    std::cout << e2.i << std::endl;
    Example::i++;
    std::cout << e1.i << std::endl;
    std::cout << e2.i << std::endl;
    return 0;
}
```

```
0
0
1
1
2
2
3
3
0
1
1
2
2
3
3
3
Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

Cuvântul cheie 'static'

4. În cadrul unei clase, **o funcție membră** declarată `static` înseamnă că toate instanțele clasei vor partaja respectiva funcție.
- O funcție membră statică nu poate accesa membrii non-statici ai instanței, deoarece nu are acces la pointerul `this`.
 - Funcțiile membre statice pot fi apelate fără o instanță a clasei.
 - Pentru a modifica datele membre non-statice ale clasei, trebuie transmis ca parametru un pointer la un obiect sau o referință din clasa respectivă.

Cuvântul cheie 'static'

```
#include<iostream>
```

```
class Example{  
public:
```

```
    int i;
```

```
    static int change_i(Example e) {
```

```
        return e.i++;
```

```
    }
```

```
};
```

```
int main() {
```

```
    Example e1;
```

```
    e1.i=5;
```

```
    std::cout << Example::change_i(e1) << std::endl;
```

```
    std::cout << Example::change_i(e1) << std::endl;
```

```
    std::cout << Example::change_i(e1) << std::endl;
```

```
    return 0;
```

```
}
```

← Parametrul este o instanță a clasei

```
5  
5  
5
```

```
Process returned 0 (0x0)   execution time : 0.125 s  
Press any key to continue.
```

Cuvântul cheie 'static'

```
#include<iostream>
```

```
class Example{  
public:
```

```
    int i;
```

```
    static int change_i (Example &e) {
```

```
        return e.i++;
```

```
    }
```

```
};
```

```
int main() {
```

```
    Example e1;
```

```
    e1.i=5;
```

```
    std::cout << Example::change_i (e1) << std::endl;
```

```
    std::cout << Example::change_i (e1) << std::endl;
```

```
    std::cout << Example::change_i (e1) << std::endl;
```

```
    return 0;
```

```
}
```

Parametrul este o referință către o instanță a clasei

```
5  
6  
7
```

```
Process returned 0 (0x0)   execution time : 0.035 s  
Press any key to continue.
```


Cuvântul cheie 'static'

```
#include<iostream>
```

```
class Example{  
public:
```

```
    int i;
```

```
    static int change_i (Example *e) {
```

```
        return e->i++;
```

```
    }
```

```
};
```

```
int main() {
```

```
    Example e1;
```

```
    e1.i=5;
```

```
    std::cout << Example::change_i (&e1) << std::endl;
```

```
    std::cout << Example::change_i (&e1) << std::endl;
```

```
    std::cout << Example::change_i (&e1) << std::endl;
```

```
    return 0;
```

```
}
```

Parametrul este un pointer către o instanță a clasei

Apelul funcției se face cu adresa instanței ca argument

```
5  
6  
7
```

```
Process returned 0 (0x0)   execution time : 0.035 s  
Press any key to continue.
```

Cuvântul cheie 'static'

```
#include<iostream>
```

```
class Example{  
public:  
    static int s;  
    static void display_s(){  
        std::cout << s << std::endl;  
    }  
};
```

Funcția statică poate accesa doar date membre statice

```
int Example::s = 6;
```

Definire dată membră statică în domeniul fișierului


```
int main(){  
    Example e1;  
    Example::display_s();  
    Example::s++;  
    e1.display_s();  
  
    return 0;  
}
```

Funcția statică poate fi apelată atât folosind numele clasei, împreună cu operatorul de rezoluție, cât și dintr-o instanță a clasei.

The image shows the word "FRIENDS" in its iconic white, hand-drawn font against a dark blue background. Each letter is separated by a small colored dot (red, blue, yellow, red, yellow, blue). An asterisk is placed at the end of the word.

***Nu *acei* prieteni**

Cuvântul cheie `friend`

 `def` `friend` = cuvânt cheie pentru a crea funcții sau clase “prietene” pentru o clasă anume.

“Prieteni” pot accesa datele membre `private` și `protected` (pe lângă cele `public`) ale unei clase => excepție de la mecanismul “ascunderii datelor”.

Funcții friend

O funcție **non-membră** a unei clase poate accesa datele și funcțiile `private` și `protected` ale unei clase dacă este declarată “prietină” pentru acea clasă.

```
#include <iostream>
class Geam{
private: ←————— Dată membră de tip private
    bool deschis;
public:
    Geam() : deschis(false) {}

    friend void change_state(Geam g, bool stare); ← Declarare funcție ca fiind “prietină” pentru clasa Geam
};
                                                ← Cuvântul cheie friend apare doar în interiorul clasei

void change_state(Geam g, bool stare){ ← Definire funcție non-membră a clasei Geam
    std::cout << g.get_deschis() << std::endl;
    g.deschis = stare;
    std::cout << g.get_deschis() << std::endl;
}

int main(){
    Geam g1;
    change_state(g1, true);
    return 0;
}
```

Class friend

O clasă “prietenă” poate accesa membrii (datele și funcțiile) `private` și `protected` ai unei clase în care este declarată prietenă.



```
#include <iostream>
class Square {
    friend class Rectangle;
    int side;
public:
    Square (int a) : side(a) {}
};
class Rectangle {
    int width, height;
public:
    int area() {return (width * height);}
    void convert (Square a) {
        width = a.side;
        height = a.side;
    }
};
int main () {
    Rectangle r;
    Square sqr (4);
    r.convert(sqr);
    std::cout << r.area(); // afiseaza 16
    return 0;
}
```

Declarare clasă ca fiind “prietenă” pentru clasa Square

Cuvântul cheie `friend` apare doar în interiorul clasei

Clasa Rectangle poate accesa toți membrii clasei Square pentru că îi este “prietenă”

Proprietăți friend

- Relația de prietenie nu este reciprocă: A îl declară prieten pe B  B îl declară prieten pe A. Acest lucru trebuie specificat în mod explicit de către B.
- Relația de prietenie nu este tranzitivă: A îl declară prieten pe B și B îl declară prieten pe C  A îl declară prieten pe C. Acest lucru trebuie specificat în mod explicit de către A.

Sfârșit capitol 2