



UNIVERSITATEA BABEŞ-BOLYAI  
Facultatea de Matematică și Informatică



# Programare orientată obiect

---

Curs 03

Laura Dioșan

# POO

---

- Elemente de bază ale limbajului C++
  - Fișiere de IO
  
- Clase
  - Declarare
  - Constructor/destructor
  - Metode
  - Utilizare (static și dinamic)
  - Clase ca și date membre
  - UML

# Fișiere de Intrare/Ieșire

---

- Procesarea fișierelor la 2 nivele:
  - Nivel inferior
    - Acces direct la SO
  - Nivel superior
    - Utilizarea unor funcții speciale de intrare-ieșire

# Procesarea fișierelor

---

- ❑ Deschiderea unui fișier
- ❑ Citirea dintr-un fișier
- ❑ Scrierea într-un fișier
- ❑ Accesarea unui fișier
- ❑ Închiderea unui fișier

# Procesarea la nivel superior

---

- header `<fstream>`
  - `ifstream`
  - `ofstream`
  
- a se consulta exemplul din
  - `03/files/appFiles.cpp`

# POO

---

- Dezvoltarea proiectării sistemelor care presupune utilizarea:
  - obiectelor – unități de bază, fiecare obiect fiind o instanță a unei clase
  - clase – relaționate prin relații de compoziție și moștenire

# POO – concepte de bază

---

## □ clasă

- definește în mod abstract caracteristicile unui lucru:
  - caracteristici (atribute, câmpuri, proprietăți)
  - comportament (metode, operații, trăsături)
- implementarea unui TAD

## □ obiect

- instanță (obiectul actual creat în timpul rulării) a unei clase
- variabilă de tip clasă

## □ metodă

- modalitate de comunicare între obiecte
- toate metodele formează interfața unui obiect

# Dezvoltarea TAD-urilor în C++

---

## □ TAD

- Exportarea unui nume (un tip de date)
- Definirea unui domeniu de valori pentru date
- Definirea unei interfețe (operațiile TAD-ului)
- Restricționarea accesului la componentele TAD-ului (acces doar prin intermediul operațiilor)
- Ascunderea implementării unui TAD



# POO - caracteristici

---

## □ Încapsulare

- capacitatea de a comprima datele cu operațiile
- ascunderea implementării → controlul accesului

## □ Moștenire

- Reutilizarea codului (folosirea și îmbunătățirea codului unei clase)

## □ Polimorfism

- Permite un comportament adaptat contextului

# Definirea clasei

---

- Similar unei date de tip **struct**:

```
class <nume_clasă> {  
    [<protecție>:] listă_membri;  
};
```

- Protecție:

- **private**

- Nu se permite nimănui accesul la membri din afară clasei
- Excepție: clasele **friend** cu clasa X pot accesa datele private ale lui X

- **protected**

- Nu se permite nimănui accesul la membri din afară clasei
- Excepție: clasele derivate din clasa X pot accesa datele protected ale lui X

- **public**

- Se permite oricui accesul la membri din afară clasei

- Lista de elemente membre

- date membre
- metode membre

# Definirea clasei

---

## □ Membri

- date
- metode

```
class <class_name>{  
    [<protection>:]  
        <type> <identif>;  
  
    [<protection>:]  
        <return_type> <fc_name>(<lfp>);  
};
```

# Clase – definirea metodelor

- imediat, în interiorul clasei → metode *inline*
  - definiție completă (antet + corp)
  - pentru metodele simple, fără bucle
  
- în afara clasei -> prin utilizarea operatorului de rezoluție/scop **::**
  - antetul metodei în interiorul clasei
  - corpul metodei în afara clasei

```
class Flower{
private:
    char* name;
    int price;
public:
    int getPrice(){
        return this->price;
    }
};
```

```
class Flower{
private:
    char* name;
    int price;
public:
    int getPrice();
};

int Flower::getPrice(){
    return this->price;
}
```

# Obiecte

## □ Declararea obiectelor

```
<class_name> <identif>;
```

- Alocarea și inițializarea unui obiect
  - la declarare
  - prin apelul **automat** al **constructor**-ului
- De-alocarea
  - la sfârșitul programului/funcției
  - prin apelul **automat** al **destructor**-ului

## □ Utilizarea obiectelor

- Componentele clasei sunt accesate:
  - direct – în interiorul metodelor membre ale clasei
  - prin folosirea operatorului "." – în alte metode (care nu sunt membre); operatorul este precedat de un obiect
  - prin folosirea operatorului "->" – operatorul este precedat de o referință a unui obiect
  - prin folosirea operatorului "::" – în alte metode (care nu sunt membre); operatorul este precedat de numele unei clase

```
class Flower{
private:
    char* name;
    int price;
public:
    static int no;

    char* getName(){
        return this->name;
    }
    int getPrice();
};

int Flower::getPrice(){
    return price;
}

int Flower::no = 0;

void main(){
    Flower f;
    cout << f.getPrice();
    cout << Flower::no;
}
```

# Constructorul unei clase

---

- Constructor – o metodă publică specială:
  - numele metodei = nume\_clasă
  - fără tip de return (nici măcar **void**)
  - scop:
    - rezervă spațiu pentru datele membre
    - inițializează datele membre
  - apelat **automat** la declararea unui obiect
  
- Destructor – o metodă publică specială:
  - numele metodei = ~nume\_clasă
  - fără tip de return (nici măcar **void**)
  - scop:
    - eliberează spațiul ocupat de datele membre
  - apelat **automat** la sfârșitul domeniului de vizibilitate
    - la sfârșitul programului → pentru obiectele globale
    - la sfârșitul funcției → pentru obiectele locale

# Constructorul unei clase

---

- Fiecare clasă trebuie să conțină cel puțin 2 constructori:
  - constructor implicit
  - constructor de copiere
  
- Alți constructori posibili
  - constructor cu parametri
  - constructor de conversie

# Constructor implicit

- fără argumente sau cu argumente implicite
- rezervă spațiu pentru datele membre
- inițializează datele membre cu valori implicite
- este apelat automat la declararea unui obiect

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower(){
        cout << "implicit constructor" << endl;
        name = new char[10];
        price = 0;
    }

    Flower(int p = 0){
        cout << "implicit constructor 2" << endl;
        name = new char[10];
        price = p;
    }
};
```

```
class <class_name>{
    ...
    public:
        <class_name>();
        <class_name>(<implicit_arg>);
    ...
};
```



# Constructor general (cu parametri)

- ❑ cu argumente
- ❑ rezervă spațiu pentru datele membre
- ❑ inițializează datele membre cu valori date

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower() { ... }
    Flower(int p = 0) { ... }

    Flower(char* n, int p){
        cout << "construtor with param" << endl;
        name = new char[strlen(n) + 1];
        strcpy(name, n);
        price = p;
    }
}
```

```
class <class_name>{
...
public:
    <class_name>(<type1 arg1>, <type2 arg2>, ...);
...
};
```

# Constructor de copiere

- ❑ crează o copie a unui obiect
- ❑ este apelat când:
  - un obiect este transmis ca parametru prin valoare
  - un obiect este returnat prin valoare
  - se inițializează un obiect cu un alt obiect

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower() { ... }
    Flower(int p = 0) { ... }
    Flower(char* n, int p) { ... }

    Flower(Flower &f){
        cout << "copy construtor" << endl;
        if (f != NULL){
            name = new char[strlen(f.name) + 1];
            strcpy(name, f.name);
            price = f.price;
        }
    }
};
```

```
class <class_name>{
...
public:
    <class_name>(<class_name> &o);
    <class_name>(<class_name> &o, <impl_arg>);
    <class_name>(<const <class_name> &o);
    <class_name>(<const <class_name> &o, <impl_arg>);
...
};
```

# Constructor de copiere

---

- ❑ crează o copie a unui obiect
- ❑ este apelat când:
  - un obiect este transmis ca parametru prin valoare
  - un obiect este returnat prin valoare
  - se inițializează un obiect cu un alt obiect
- ❑ dacă nu există, compilatorul produce un constructor de copiere implicit care realizează o copie bit cu bit a obiectului curent → probleme cu datele alocate dinamic
- ❑ alternative ale constructorului de copiere
  - prevenirea transmiterii prin valoare

# Constructor de conversie

- construiește un obiect de tipul clasei curente dintr-un obiect/variabilă de alt tip

```
class <class_name>{  
    ...  
    public:  
        <class_name>( <other_type> &o);  
        <class_name>( <other_type> &o, <impl_arg>);  
        <class_name>( const <other_type> &o);  
        <class_name>( const <other_type> &o, <impl_arg>);  
    ...  
};
```

```
class Plant{  
private:  
    char* plant;  
    int value;  
public:  
    Plant();
```

```
class Flower{  
private:  
    char* name;  
    int price;  
public:  
    Flower() { ... }  
    Flower(int p = 0) { ... }  
    Flower(char* n, int p) { ... }  
    Flower(Flower &f) { ... }
```

# Destructor

- Eliberează resursele de memorie alocate de constructor
  - memoria alocată în heap
  - conexiunile cu baze de date
  - conexiunile în rețea
  - fișierele deschise
- Este unic
- Nu returnează nimic
- Nu are parametri

```
class <class_name>{  
    ...  
    public:  
        ~<class_name>();  
    ...  
};
```

```
class Flower{  
private:  
    char* name;  
    int price;  
public:  
    Flower() { ... }  
    Flower(int p = 0) { ... }  
    Flower(char* n, int p) { ... }  
    Flower(Flower &f) { ... }  
    Flower(Plant &p) { ... }  
  
    ~Flower(){  
        cout << "destrutor" << endl;  
        if ( name != NULL){  
            delete[] name;  
            name = NULL;  
        }  
    }  
}
```

# Forma standard a unei clase

---

## □ Date

- attributele clasei

## □ Metode

- Constructor implicit
- Constructor de copiere
- Destructor
- Operator de atribuire (a se consulta cursul următor)
- Alte metode

# Obiecte ca și parametri

- Dacă obiectul nu-și schimbă valoarea în interiorul funcției, el va fi apelat ca parametru **const**

```
Flower::Flower(const Flower &f){
    cout << "copy constructor" << endl;
    name = new char[strlen(f.name) + 1];
    strcpy(name, f.name);
    price = f.price;
}
```

```
void change(Flower& f){
    f.setName("daisy");
}
```

- Dacă o metodă nu modifică obiectul apelant, metoda va fi **const**

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower();
    Flower(char* name);
    Flower(const Flower& f);
    ~Flower();

    char* getName() const;
    return name;
}
```

# Obiecte ca și parametri

- Obiectele sunt transmise prin referință pentru a evita apelarea constructorului de copiere

```
void withRef(Flower &f){
    char* s = f.toString();
    cout << s << endl;
    if (!s){
        delete[] s;
        s = NULL;
    }
}

void withoutRef(Flower f){
    char* s = f.toString();
    cout << s << endl;
    if (!s){
        delete[] s;
        s = NULL;
    }
}
```

- Un obiect de ieșire este returnat prin **valoare** când el a fost creat ca obiect temporar în interiorul funcției

```
Flower newFlower()
{
    Flower f("dais");
    return f;
}

Flower& newFlowerR
```

```
void main(){
    Flower f1 = newFlower();
    cout << f1.toString();
    Flower f2 = newFlowerR
```



# Exemplu – clasă simplă

---

- Clasa Flower
  - Date membre:
    - Name
    - Price
  - Metode:
    - Constructors
    - Destructor
    - Set-ers
    - Get-ers
    - toString
    - Compare
- a se consulta exemplul din
  - 03/class

# Exemplu – o clasă mai complexă

- ❑ Este nevoie de un constructor implicit pentru clasa *Flower*
  - Dacă clasa *Flower* nu are un constructor implicit, compilatorul va sintetiza unul
  - Dacă clasa *Flower* are alte tipuri de constructori → erori

```
class Flower{
private:
    char* name;
    int price;
public:
    Flower(char* name, int p);
    Flower(const Flower &f);
    Flower(char* s);
    //Flower(Plant &p);
    ~Flower();
    char* getName();
    int getPrice();
    void setName(char* n);
    void setPrice(int p);
    char* toString();
    bool compare(Flower &f);
};
```

```
class Gardener{
private:
    char* name;
    Flower f;
public:
    Gardener();
    Gardener(char* n, Flower& f);
    Gardener(char* s);
    Gardener(const Gardener &g);
    ~Gardener();
    void setName(char* n);
    void setFlower(Flower &f);
    char* getName();
    Flower& getFlower();
    char* toString();
    bool compare(Gardener g);
};
```

```
void main(){
    Gardener g; //error
}
```

# Temă

---

- Scrieți un program pentru a defini și utiliza:
  - clasa Student (PIN, nume, vârstă) și
  - scrieți un program de test pentru această clasă
  
- Scrieți un program pentru a defini și utiliza:
  - clasa Punct(x, y),
  - clasa Segment (punct1, punct2) și
  - clasa Figură (nrDeSegmente, segmente).
  - Scrieți un program de test pentru aceste clase

# Cursul următor

---

- Clase
  - Elemente prietene
  - Supraîncărcarea operatorilor