

Programare II

Limbajul C/C++

CURS 8



CUPRINS

- ❑ Membri statici ai claselor
- ❑ Membri prieteni ai claselor
- ❑ Modificatori de acces
- ❑ Relații între clase

Membri statici

- ❑ Date/funcții prefixate cu specificatorul static
 - ❑ Datele statice există într-o singură copie comună tuturor obiectelor
 - ❑ Funcțiile statice realizează operații care nu sunt asociate obiectelor individuale, ci întregii clase
- ❑ Accesare
 - ❑ Indicarea numelui clasei și folosirea operatorului de rezoluție (`X::fctStatica();`)
 - ❑ Specificând obiectul clasei și folosind operatorii de selecție (`X x; x.fctStatica();`)

Câmpuri statice

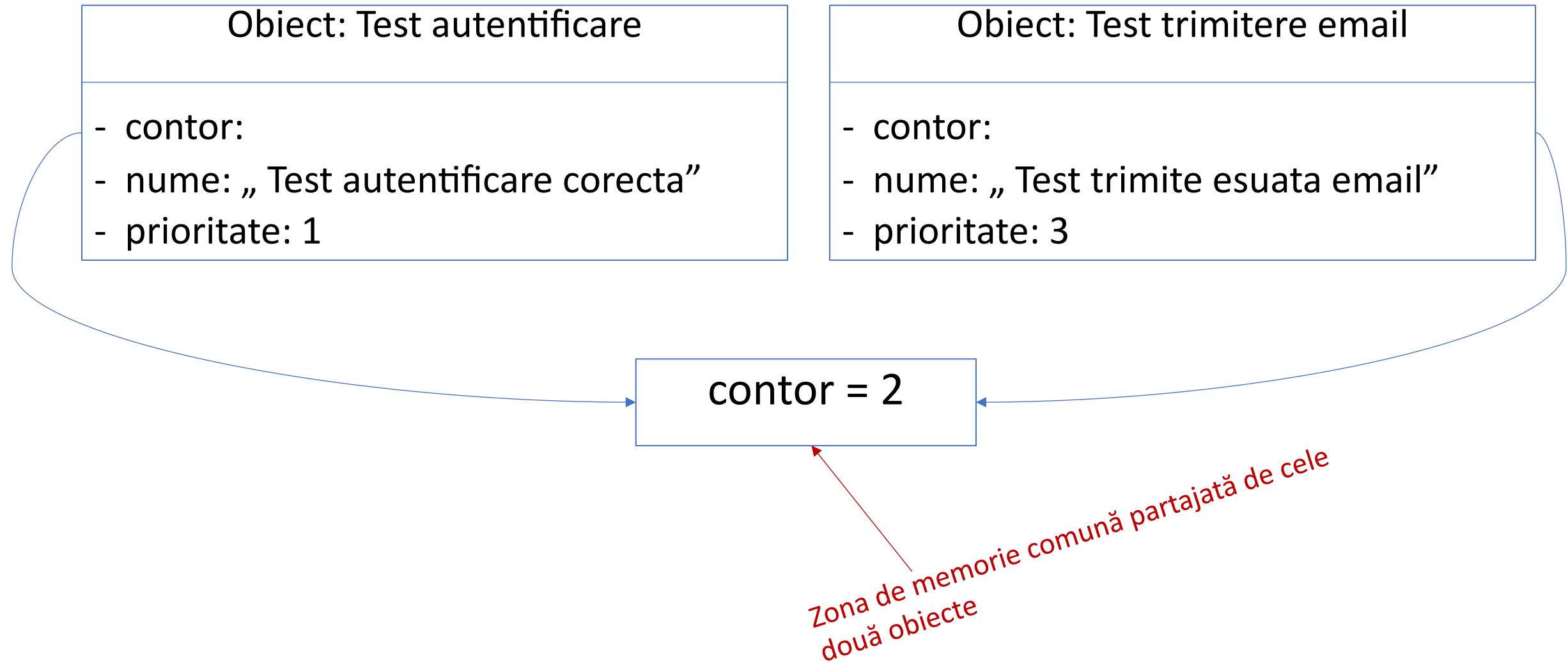
- ❑ Alocarea memoriei și inițializarea se realizează separat
- ❑ Se inițializează cu zero înainte să se creeze primul obiect
- ❑ Au aceleași proprietăți ca și variabilele globale doar că sunt limitate de domeniul de vizibilitate al clasei
- ❑ Exemplu

```
class Test { ...  
    static int contor;  
    char * nume;  
    int prioritate;  
    Test () { contor++; } ...  
};  
int Test::contor = 10;
```

Declarare variabilă membru statică

Inițializare variabilă membru statică

Câmpuri statice



Functii statice

❑ Proprietăți

- ❑ Pot accesa direct membri statici ai clasei
- ❑ Nu pot accesa pointerul `this` (membri nestatici ai clasei)

❑ Restricții

- ❑ Nu pot fi **virtuale**
- ❑ Nu pot fi declarate `const` sau `volatile`

❑ Transformare apel de către compilator

- ❑ Apel funcție membră: `foo(7);` → `foo(this, 7)`
- ❑ Apel funcție statică: `fooStatica(7);` → `fooStatica(7)`

La fel ca în Python, la funcțiile membre clasei primul parametru este o referință la obiectul curent

Functii statice. Exemplu

❑ Exemplu

❑ Genereare ID-ului pentru o persoana

```
class Persoana{  
    ...  
    static int urmatorulId;  
    int id;  
public:  
    ...  
    static int urmatorul_id_utilizator();  
};  
Persoana::Persoana (char *n, int an) {  
    this->id =  
        Persoana::urmatorul_id_utilizator();  
}
```

Pentru claritate la apelul în interiorul clasei este bine să fie prefixată de numele clasei

```
int Persoana::urmatorul_id_utilizator() {  
    urmatorulId++;  
    return urmatorulId;  
}  
  
int Persoana::urmatorulId = 0;
```

Initializare variabilă statică

La definirea funcției în exteriorul cheie static mai adaugă cuvântul cheie static

Functii prietene

❑ Funcție prietenă (friend)

- ❑ Funcție care are **acces la membri non-publici** ai unei clase
- ❑ Prefixate de cuvântul cheie **friend**
- ❑ **Prototipul** este specificat **în definiția clasei**
- ❑ **Definiția** este specificată **în afara domeniului clasei**

❑ Sintaxă

```
class X { ...  
    friend tipReturn numeFunctie( ... );  
    ...  
};  
tipReturn numeFunctie( ... ) {  
    ...  
}
```

Prototip funcție

Definiție funcție

Functii prietene. Exemplu

- ❑ Funcție de calcul al modulului unui număr complex

- ❑ Exemplu

```
class Complex {  
    private:  
        int re, im;  
    public:  
        friend double modul(const Complex & c);  
};  
double modul(const Complex & c){  
    return sqrt(c.re * c.re + c.im * c.im);  
}
```

- ❑ Apel

```
Complex c(6, 7);  
modul(c);
```

Clase prietene

❑ Accesul este unidirecționat

❑ Dacă B este prietenă cu A, B poate accesa membri non-publici a lui A, dar A nu poate accesa membri non-publici a lui B

```
class ArboreBinar;  
class Nod {  
private:  
    int data;  
    int cheie;  
    ....  
    friend class ArboreBinar;  
};  
class ArboreBinar {  
private:  
    Nod *radacina;  
public:  
    int gasesteCheie(int);  
};
```

```
int ArboreBinar::gasesteCheie(int val) {  
    if (val == radacina->cheie) {  
        return radacina->data;  
    }  
    //restul cautarii  
}
```

Declarare parțială a clasei Arbore binar,
deoarece referința dintre clase este circulară

Acces la câmpurile private ale
clasei Nod din clasa ArboreBinar

Funcții/clase prietene

- ❑ Utilitate
 - ❑ Oferă un **acces mai eficient** la membri decât în cazul apelului de funcții
 - ❑ Suprâncărcarea operatorilor – un acces mai ușor la datele private
- ❑ Funcțiile/clasele prietene au acces la **toți** membri clasei, ceea ce **violează încapsularea** datelor → atenție la utilizare
- ❑ Funcțiile/clasele membre pot modifica starea unei clase.
 - ❑ Recomandare: folosirea funcțiilor membre pentru a modifica date

Modificatori de acces

- ❑ const
- ❑ mutable

Modificatorul const

❑ Date constante

- ❑ Nu pot fi modificate
- ❑ Utile pentru declararea de constante

❑ Inițializarea

- ❑ În lista de inițializări a constructorului clasei

❑ Sintaxă

- const variabilă

```
class Pagina {  
    // declarare  
    const int ci ;  
    static const int MAX_VIEWS;  
  
public:  
    Pagina () : ci(17) {  
        // initializarea unui membru const  
        // în interiorul unei liste de inițializare  
    }  
  
};  
// initializarea unui membru  
//static constant  
const int Pagina ::MAX_VIEWS = 256;
```

Modificatorul const

❑ Funcții constante

- ❑ Nu pot modifica starea unui obiect
- ❑ Cod mai clar
- ❑ Împiedică modificările accidentale ale datelor

❑ Sintaxă

```
tipDeReturnnumeFunctie (tipVar [,  
tipVar]) const;
```

```
class Utilizator{ private:  
    int id;  
    char * username;  
public:  
    int getId() const {  
        id=0;  
        return id;  
    }  
    char * getUsername() const;  
};  
char * Utilizator::getUsername () const  
{  
    return username;  
}
```

Modificatorul const

- ❑ Parametrii constanți ai funcțiilor

- ❑ Un parametru constant nu își poate modifica valoarea în interiorul funcției

```
class Pagina {  
    int nrIntroduceriParolaGresita;  
public:  
    Pagina( const Pagina & );  
};  
Pagina :: Pagina( const Pagina &p) {  
    //eroare  
    p.nrIntroduceriParolaGresita++;  
}
```

Modificatorul mutable

- ❑ Se aplică la membri date
❑ Pot fi întotdeauna modificați chiar și în funcții constante
❑ Utili pentru membri care trebuie modificați în funcții const și nu reprezintă interes pentru starea internă a obiectului
- ❑ Sintaxă
 - ❑ mutable numeVariabilă;

```
class Utilizator{  
    private:  
        mutable int nraccessari;  
  
    public:  
        int getId() const {  
            nraccessari++;  
            return id;  
        }  
};
```

Relații între clase

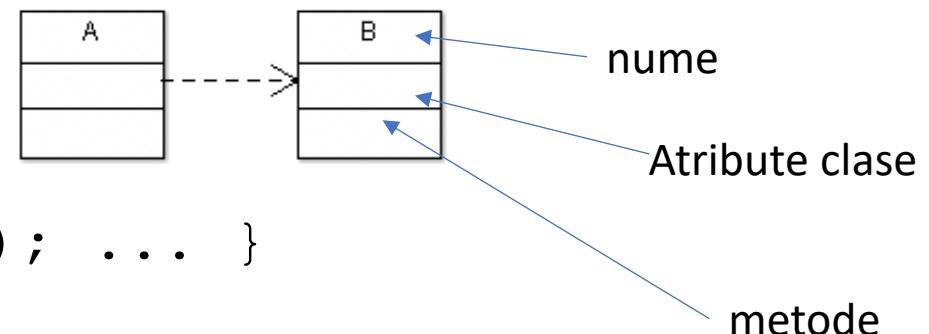
- ❑ Conceptele nu există izolate. Ele **coexistă și interacționează**.
- ❑ La elaborarea modelului obiectual al unei aplicații se disting următoarele etape:
 - ❑ **Identificarea claselor** → corespund conceptelor aplicației - substantivele –
 - ❑ **Stabilirea relațiilor dintre clase** → corespunde specificațiilor aplicației – verbe –
- ❑ Tipuri de relații
 - ❑ Asociere o relație în care obiectele unei clase știu de obiectele altei clase (has-a)
 - ❑ **Dependență** o relație în care obiectele unei clase știu de obiectele altei clase (use-a)
 - ❑ **Agregare** o relație parte întreg (is-part-of)
 - ❑ **Compoziție** este similară cu asocierea dar mai strictă (contains)
 - ❑ **Moștenire** (specializare) este o relație de generalizare - specializare (is-a, kind-of)

Relația de dependentă

- ❑ Se identifică prin „uses a”
 - ❑ O funcție a unui obiect apelează o funcție membră a altui obiect sau are nevoie de o instanță a unei clase pentru a realiza o acțiune

❑ Exemple de implementare

```
❑ class B { ... };  
❑ class A { public:  
❑ void method1(B b) { // . . . }  
    void method2() { B tempB = new B(); ... }  
❑ };
```



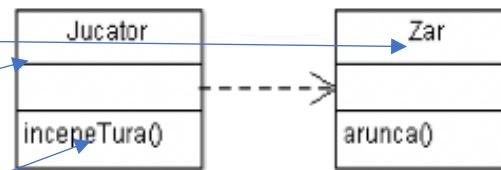
- ❑ OBS: Clasa A nu va conține o variabilă membru de tipul clasei B

Relația de dependentă

❑ Exemplu

- ❑ Un jucător de „Nu te supara frate” folosește un zar

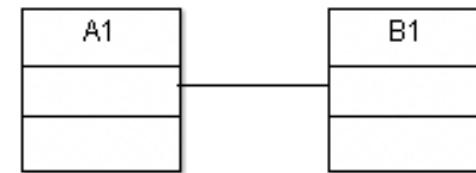
```
class Zar {  
    public void arunca() { ... }  
}  
class Jucator {  
    public void incepeTura(Zar zar) {  
        zar.arunca();  
        ...  
    }  
}
```



Asociere

- ❑ Relație de cooperare între clase - corespunzătoare unui verb oarecare din specificație
- ❑ Obiectele au propriul ciclu de viață și nu există proprietar
- ❑ Este identificată prin „**has-a**”
- ❑ Implementare: variabile membru (pointeri sau referințe) la obiectele asociate

```
class B1 { ... };  
class A1 {  
private:  
    B1 *b1;  
public:  
    B1& getB1() const { return *b1; }  
};
```



- ❑ OBS: greu de întreținut

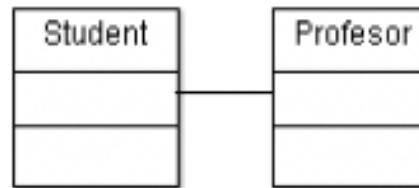
Asociere exemplu

□ Clasele Student și Profesor

- un Student poate fi asociat la mai mulți Profesori
- un Profesor poate fi asociat la mai mulți Studenți
- nu există relație de apartenență între obiecte
- ambele pot fi create și sterse independent

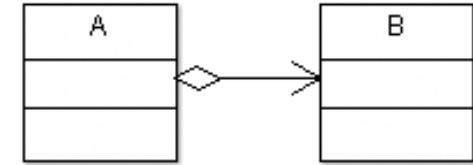
□ Implementare

```
class Profesor {  
    list<Student*> studentiLicenta;  
public:  
    void adaugare(Student &s) { studentiLicenta.push_back(&s); }  
    ~Profesor() {}  
};  
class Student{  
    Profesor *profesorCoordonator;  
public:  
    void setProfesorCoordonator(Profesor *p) { profesorCoordonator = p; }  
    ~Student () {}  
};
```



Agregare

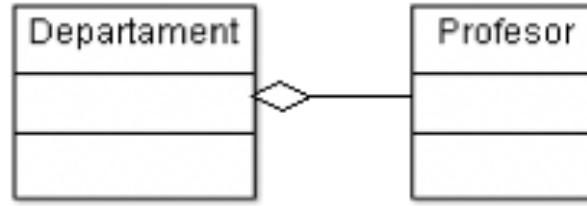
- ❑ Un caz special de relație de asociere
 - ❑ Fiecare obiect are propriul ciclu de viață
 - ❑ Un obiect "copil" poate apartine doar unui obiect "părinte" nu mai multor obiecte
-
- ❑ Exemplu
 - ❑ casele Profesor și Departament
 - ❑ un profesor nu poate apartine unui singur departament, nu poate apartine la mai multe
 - ❑ dacă stergem un departament, obiectul de tip profesor nu va fi distrus



Agregar

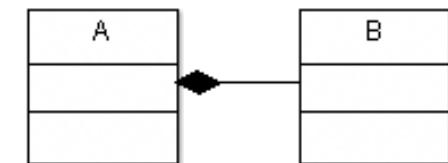
□ Implementare

```
class Departament{  
    list<Proferor*> profesori;  
public:  
    void addProfesor(Profesor *p) { profesori.push_back(p); }  
    ~Departament() {}  
};  
class Profesor {  
    Departament *dept;  
public:  
    Profesor(const Departament *d) { dept=d; }  
    ~Profesor() {}  
};
```



Compoziție

- ❑ Subiectele aggregate aparțin exclusiv aggregatului din care fac parte, iar durata de viață coincide cu cea a aggregatorului
- ❑ Clasa copil nu poate exista decât dacă există o instanță a clasei părinte.
- ❑ În exemplul de mai jos instanța clasei Comisie există atât timp cât există instanța clasei
- ❑ Exemplu
 - ❑ clasele Casă și Cameră
 - ❑ O casă poate conține mai multe camere
 - ❑ Nu poate exista o cameră care să nu fie atașată unei case



Compoziție

Implementare

```
class Camera {  
    public Camera( int, int, int);  
};  
  
class Casa {  
    list<Camera *> camere;  
public:  
    void addcamere( int lungime, int latime, int inaltime) {  
        camere.push_back( new Camera(lungime, inaltime, latime));  
    }  
~Casa() {  
    for (int i=0; i<camere.size(); i++)  
        delete camere[i];  
}  
};
```

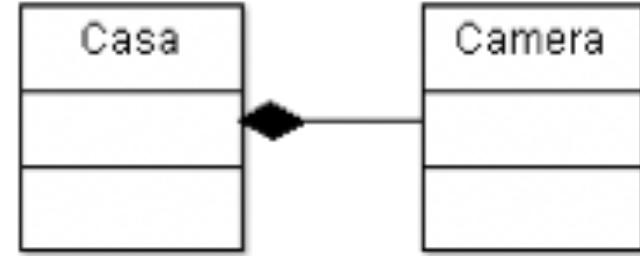


Diagrama de clase

Separarea codului

- ❑ Fișiere sursă care conțin implementarea codului sursă
 - ❑ Extensie .cpp
 - ❑ Implementarea clasei
 - ❑ Programul principal
 - ❑ Programe de test
 - ❑ Implementarea funcțiilor membre
 - ❑ Într-un fișier sursă separat de definiția clasei
 - ❑ Folosirea operatorului de definire a scopului :: pentru a lega definiția de declarația funcției
 - ❑ Detaliile de implementare sunt ascunse
 - ❑ Codul clientului nu trebuie să știe detaliile de implementare
- ❑ Fișierele header
 - ❑ Au extensia .h
 - ❑ Fișiere care conțin declararea clasei
 - ❑ Permit compilatorului să recunoască clasele când sunt folosite în alt loc
 - ❑ Interfețe
 - ❑ Descriu serviciile pe care clienți unei clase le pot folosi și modul în care se obțin aceste servicii

Exemplu

- Crearea clasei Color care are ca atribute cele 3 culori fundamentale red, green, blue.
- Declararea clasei in fisier .h

```
#ifndef COLOR_H_INCLUDED
#define COLOR_H_INCLUDED
class Color{
    int red, green, blue;
public:
    Color(int=0, int=0, int=0);
    void display();
private:
    double validateColor(double);
};
#endif // COLOR_H_INCLUDED
```

Care este rolul directivelor de precompilare?

Asigură unicitatea declarării clasei Color într-un proiect



ÎNTREBĂRI