Clase II

Curs anterior

- Clase
- Specificatori de acces
- Contructorii
- Destructorii
- Constructor de copiere

Cuprins

- Cuvântul cheie this
- Funcţii/clase prietene
- Membri statici
- Modificatori de acces
- Relaţii între clase
  - Asociere
  - Agregare
  - Compoziţie
  - Moştenire

Cuprins

- Cuvântul cheie this
- Funcţii/clase prietene
- Membri statici
- Modificatori de acces
- Relaţii între clase
  - Asociere
  - Agregare
  - Compoziţie
  - Moştenire

Cuvântul cheie this

- Cuvântul cheie this – autoreferinţă
- Este un pointer accesibil doar dintr-un context non-static al unei clase, structuri sau uniuni
- Pointează spre obiectul pentru care funcţia membru este apelată
- Folosirea acestuia este implicită dar se poate utiliza şi explicit

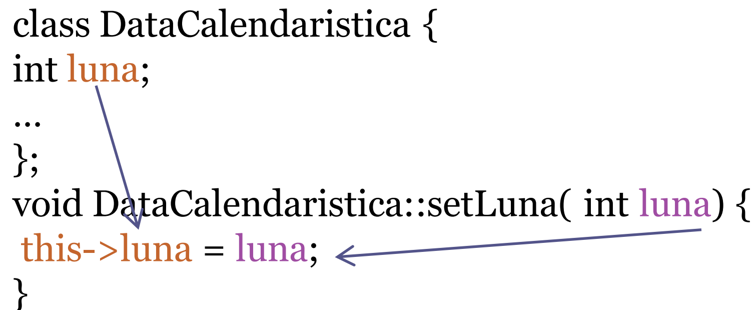
```
// declaratii echivalente
void DataCalendaristica::setLuna( int mn ) {
 luna= mn;
 this->luna = mn;
 (*this).luna = mn;
}
```

Cuvântul cheie this

- Folosire explicită
  - Identificarea obiectului curent

```
DataCalendaristica & copie(DataCalendaristica &c) {
 if (&c != this){
 ....
 }
 return *this;
}
```

  - Rezolvarea ambiguităţilor



```
class DataCalendaristica {
int luna;
...
};
void DataCalendaristica::setLuna( int luna) {
 this->luna = luna;
}
```

Cuprins

- Cuvântul cheie this
- Funcţii/clase prietene
- Membri statici
- Modificatori de acces
- Relaţii între clase
  - Asociere
  - Agregare
  - Compoziţie
  - Moştenire

Funcţii prietene

- Funcţie prietenă (friend)
  - Funcţie care are acces la membri non-publici ai unei clase
  - Prefixate de cuvântul cheie *friend*
  - Prototipul este specificat în definiţia clasei
  - Este definită în afara domeniului clasei
- Sinataxă

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

Prototip (friend tipReturn numeFunctie(…);)

Definiție (tipReturn numeFunctie(…){)

Funcţii prietene. Exemplu

- Funcţie de calcul al modulului unui număr complex

```
class Complex {
private:
  int re, im;
public:
 ...
};
```

Cum definim o funcţie de calcul al moduluil numarului complex?

Funcţii prietene. Exemplu

- Funcţie de calcul al modulului unui număr complex

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

- Apel

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

Funcţii prietene. Exemplu

- Funcţie de calcul al modulului unui număr complex

```
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 unidirecţionat
  - 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
}
```

Funcţii/clase prietene

- Utilitate
  - Oferă un acces mai eficient la membri decât în cazul apelului defuncţii
  - Supraîncărcarea operatorilor – un acces mai uşor la datele private

- Funcţiile/clasele prietene au acces la toţi membri clasei, ceea ce violează încapsularea datelor → atenţie la utilizare

- Funcţiile/clasele membre pot modifica starea unei clase.
  - Recomandare: folosirea funcţiilor membre pentru a modifica date

Cuprins

- Cuvântul cheie this
- Funcţii/clase prietene
- Membri statici
- Relaţii între clase
  - Asociere
  - Agregare
  - Compoziţie
  - Moştenire

Membri statici

- Date/funcţii prefixate cu specificatorul static
- Datele statice există într-o singură copie comună tuturor obiectelor
- Funcţiile statice realizează operaţii care nu sunt asociate obiectelor individuale, ci întregii clase
- Accesare
  - Indicarea numelui clasei şi folosirea operatorului de rezoluţie (X::fctStatica();)
  - Specificând obiectul clasei şi folosind operatorii de selecţie (X x; x.fctStatica();)

Date statice

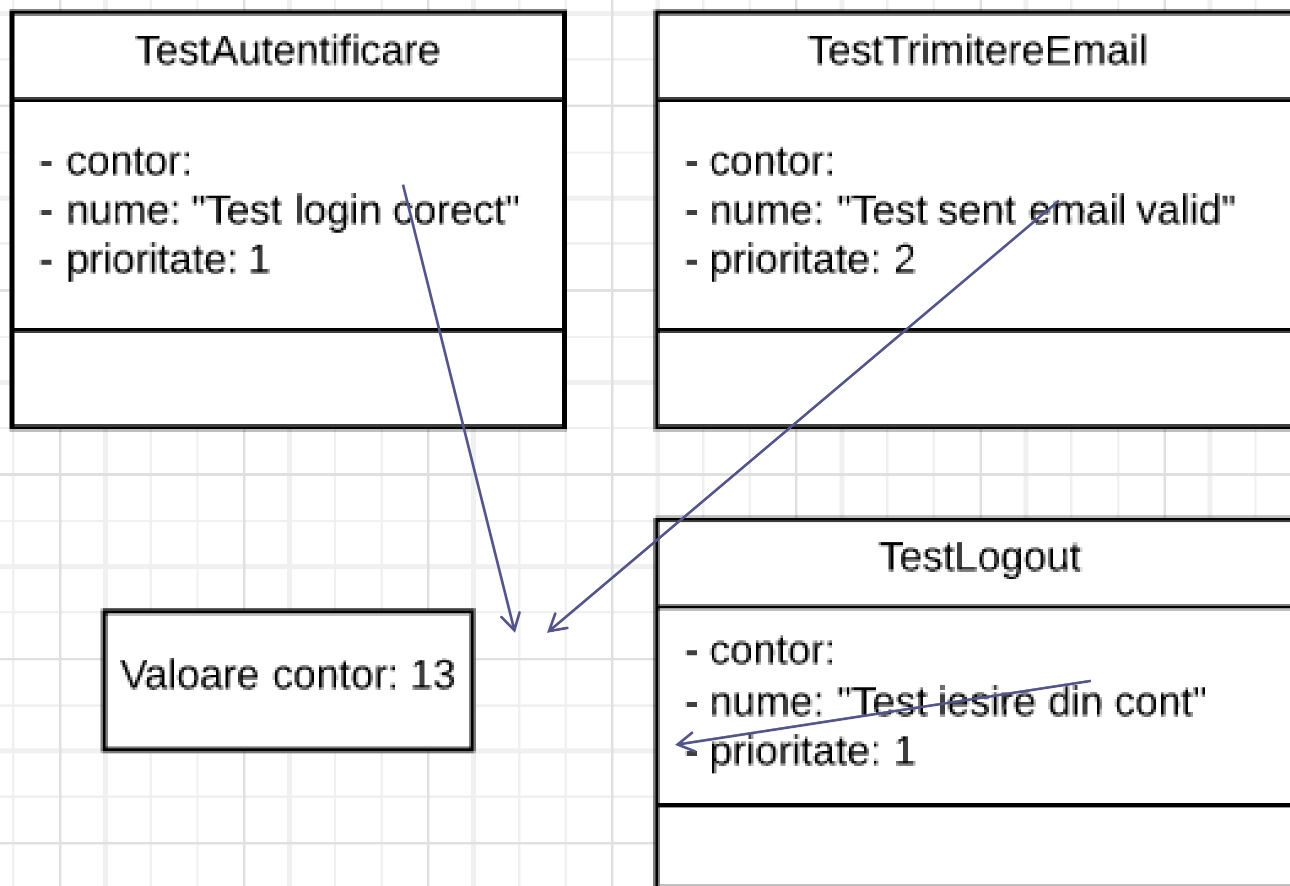
- Alocarea memoriei şi iniţializarea se realizează separat

```
class Test {
 ...
  static int contor;
  char * nume;
  int prioritate;

   Test () { contor++;}
 ...
};
int Test::contor = 10;
```

- Se iniţializează cu zero înainte să se creeze primul obiect
- Au aceleaşi proprietăţi ca şi variabilele globale doar că sunt limitate de domeniul de vizibilitate al clasei

Date statice



Funcții statice

- Proprietăţi
  - Pot accesa direct membri statici ai clasei
  - Nu au acces la pointerul this (membri nestatici ai clasei)

Transformare apel de către compilator
- Apel funcţie membră: foo (7);→ foo(this *, 7)
- Apel funcţie statică: fooStatica (7);→ fooStatica(7)

  - Nu poate exista o variantă statică şi una nestatică a aceleaşi funcţii
  - Nu pot fi virtuale
  - Nu pot fi declarate const sau volatile

Funcții statice

- Exemplu
  - Calcul ID-ului pentru un utilizator

```
class Utilizator{
  private: int id;
  static int urmatorul_id;
 public:
   static int urmatorul_id_utilizator() {
         urmatorul_id ++;
         return urmatorul_id;
   }
   Utilizator() {
         id = Utilizator:: urmatorul_id ++; //or, id = urmatorul_id_utilizator();
   }
};
 int Utilizator :: urmatorul_id = 0;
```

Cuprins

- Cuvântul cheie this
- Funcţii/clase prietene
- Membri statici
- Modificatori de acces
- Relaţii între clase
  - Asociere
  - Agregare
  - Compoziţie
  - Moştenire

Modificatori de acces

- const
- mutable
- volatile

Modificatorul const

- Date constante
  - Nu pot fi modificate
  - Utile pentru declararea de constante
  - Iniţializarea
    - În lista de iniţializări 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 iniţializare
  }
};
// initializarea unui membru
//static  constant
const int Pagina ::MAX_VIEWS = 256;
```

Modificatorul const

- Funcţii constante
  - Nu pot modifica starea unui unui obiect
  - Cod mai clar
  - Împiedică modificările accidentale ale datelor
- Sintaxă
  - tipDeReturn numeFunctie (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 dată
  - Pot fi întotdeauna modificaţi chiar şi în funcţii constante
  - Utili pentru membri care trebuie modificaţi în funcţii const şi nu reprezintă interes pentru starea internă a obiectului

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

- Sintaxă
  - mutable numeVariabilă;

Modificatorul volatile

- Indică faptul că varibile pot fi modificate din exteriorul programului (sistemul de întreruperi, operații de I/O mapate pe firul de execuție)
- Variabile nu sunt controlate de program
  - Compilatorul nu poate realiza anumite optimizări (citiri redundante – de fiecare dată când variabila este accesată compilatorul reîncarcă valoarea în memorie)
- Sintaxă
  - TipData volatile listaVariabile;
  - volatile TipData listaVariabile;
- Exemplu
  - volatile int Vint; int volatile * Vintptr;

Cuprins

- Cuvântul cheie this
- Funcţii/clase prietene
- Membri statici
- Modificatori de acces
- Relaţii între clase
  - Asociere
  - Agregare
  - Compoziţie
  - Moştenire

Relații între clase

- Conceptele nu există izolate. Ele coexistă şi interacţionează

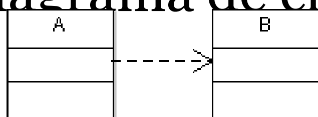
- La elaborarea modelului obiectual al unei aplicaţii se disting următoarele etape:
  - Identificarea claselor → corespund conceptelor aplicaţiei - substantivele –
  - Stabilirea relaţiilor dintre clase →corespunde specificaţiilor aplicaţiei – verbe –

- Tipuri de relaţii
  - **Asociere** o relaţie în care obiectele unei clase ştiu de obiectele altei clase (**has-a**)
    - **Dependenţă** o relaţie în care obiectele unei clase ştiu de obiectele altei clase (**use-a**)
    - **Agregare** o relaţie parte întreg (**is-part-of**)
    - **Compoziţie** este similară cu asocierea dar mai strictă (**contains**)

  - **Moştenire** (specializare) este o relaţie de generalizare - specializare (**is-a, kind-of**)

Relația de dependență

- Se identifică prin „uses a"
  - O funcţie a unui obiect apelează o funcţie membră a altui obiect sau are nevoie de o instanţă a unei clase pentru a realiza o acţiune
- Diagrama de clase



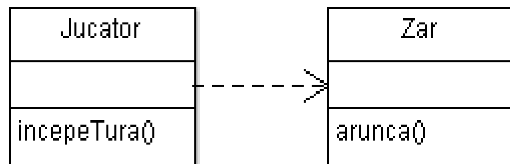
- Exemple de implementare

```
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 dependență

- Diagramă de clase

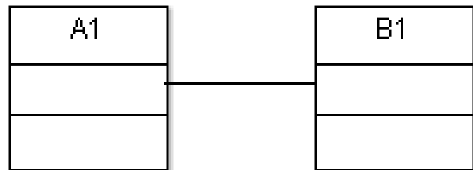


- Implementare

```
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 propietar
- Este identificată prin „has-a”

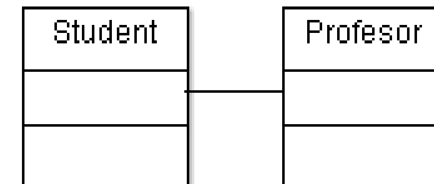


- Implementare: variabile membru (pointeri sau referinţe) la obiectele asociate

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

- OBS: greu de întreţinut

Asociere exemplu



- Clasele Student și Profesor
  - un Student poate fi asociat la mai mulți Profesori
  - un Profeso 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;
    void setProfesorCoordonator(Profesor *p) { profesorCoordonator = p; }
    ~Student () {}
};
```

Asociere

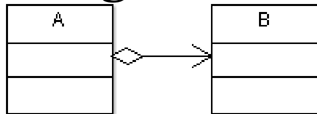
- Multiplicitate

Notație	Măsură
1	Doar unu
*	Zero sau mai multe
0..5	Zero la cinci
0..4,6,10	Zero la patru sau şase sau zece
	Exact unu (default)

- Cazuri speciale
  - agregarea
  - compoziția

Agregare

- Un caz special de relație de asociere
- Fiecare obiect are propriul ciclu de viață
- Un obiect ”copil” poate aparține doar unui obiect ”părinte” nu mai multor obiecte
- Diagrama de clase



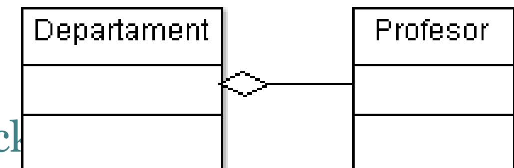
- Implementare
  - class B { … };
  - class A {

  - };

Agregare

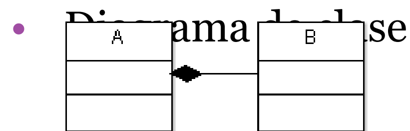
- casele Profesor și Departament
  - un profesor nu poate aparține unui singur departament, nu poate apartine la mai multe
  - dacă stergem un departament, obiectul de tip prfesor nu va fi distrus
- Implementare

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



Compoziţie

- Subobiectele agregate aparţin exclusiv agregatului din care fac parte, iar durata de viaţă coincide cu ce a agregatorului
- Clasa copil nu poate exista decât dacă există o instanţă a clasei părinte.
- În exemplul de mai jos instanţa clasei Comisie există atâta timp cât există instanţa clasei
- Diagrama de clase



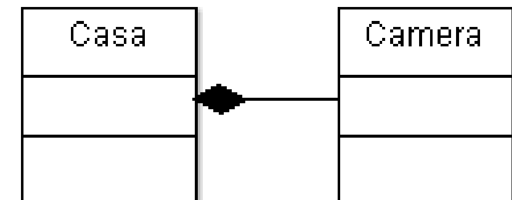
- Implementare

```
class B { … };
class A {
    B *b;
    public:
        A(){
            b = new B();
        }
        ~A(){
            delete b;
        }
}
```

Compoziție

- clasele Casă și Cameră
  - O casă poate conține mai multe camere
  - Nu poate exista o cameră care să nu fie atașată unei clase
- Implementare

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



Moştenire

- Este un mecanism care permite unei clase A să moştenescă (date şi funcţii) a unei clase B.
- Relaţie
  - Nivel de clasă kind-of (Cercul este un tip de Formă)
  - Obiect is-a (Obiectul cerc1 este o formă)
- B clasă de bază
- A subclasă