

Programare II

Limbajul C/C++

CURS 6



Curs anterior

- ❑ Programare orientată obiect. Abstractizarea datelor
- ❑ Diferențe / noutăți C – C++
 - ❑ Streamuri/operatori de IO
 - ❑ Supradefinirea numelui de funcții
 - ❑ Funcții cu parametrii impliciți
 - ❑ Alocarea dinamică a memoriei – operatorii new și delete

Curs curent

☐ Clase. Obiecte

- ☐ Specificatori de acces

- ☐ Constructori

- ☐ Destructor

Ce este un program?

- ❑ Programare structurată

 - ❑ Structuri de date + Algoritmi = Program

- ❑ Programare orientată obiect

 - ❑ Date + Metode = Obiect

Clase

❑ Ce este o clasă?

- ❑ O clasă este o implementare a unui **tip de date** (concret, abstract sau generic).
Definește **attribute** și funcții care descriu structura de date și **operațiile** care se pot efectua cu acest tip de date

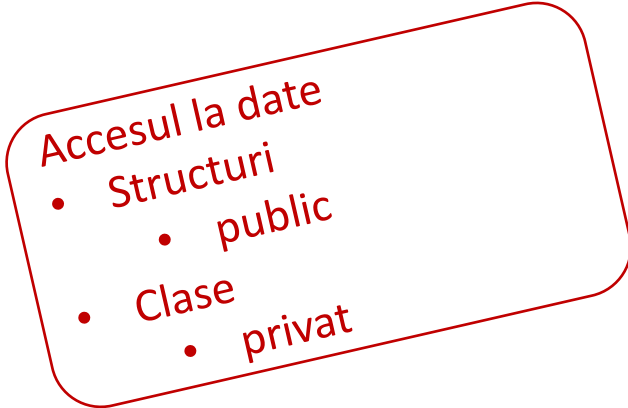
❑ Exemple

- ❑ Universitate
- ❑ Student
- ❑ Profesor
- ❑ Amfiteatru
- ❑ Sală
- ❑ etc

Clase

- ❑ Clasa este un grup de obiecte care au caracteristici comune
- ❑ Clasele în C++ pot fi definite folosind cuvintele cheie struct sau class
- ❑ Sintaxă

```
class X {  
    // variabile membru  
    // functii membru  
};  
  
struct X {  
    // variabile membru  
    // functii membru  
};
```



Accesul la date

- Structuri
 - public
- Clase
 - privat

Obiecte

- ❑ Este un element finit și particular al unui model
- ❑ Obiectele sunt create prin declararea de variabile
- ❑ Sintaxă
 - ❑ `TipDataAbstact numeVariabila;`
- ❑ Exemplu
 - ❑ `Universitate ubb;`
 - ❑ `Universitate *uvt = new Universitate;`

Obiecte

- ❑ Obiectele sunt entități create la execuție (run-time), de bază ale unui sistem orientat obiect
- ❑ Fiecare obiect are asociate date și funcții care definesc operațiile care au sens asupra obiectului
- ❑ Este o entitate care există în lumea reală
- ❑ Un obiect este o instanță a unei clase

Exercițiu

☐ Florărie

- ☐ Identificare obiectelor dintr-o florărie

- ☐ Interacțiunile dintre ele

- ☐ Proprietățile comune

- ☐ Ce clase se pot defini?



Exercițiu

☐ FLORĂRIE

- ☐ Identificare obiectelor dintr-o bibliotecă
 - ☐ Produse, accesorii, plante ghiveci, plante aranjamente, client, angajat,
- ☐ Interacțiunile dintre ele
 - ☐ Un client comandă un aranjament
 - ☐ Un angajat comandă produse lipsă
 - ☐ Un angajat folosește anumite accesorii care trebuie extrase din gestiune
- ☐ Proprietățile comune
 - ☐ Produsele care poate fi ghiveci, plante aranjamente
- ☐ Ce clase se pot defini?
 - ☐ Florărie, Client, Angajat, Produs, Accesoriu, Floare



Clase

- ❑ Caracterizează o colecție de obiecte similare

- ❑ Sintaxă

```
class numeClasă [: Listă clase de bază] {  
    Date/câmpuri/variabile membre; //informații  
    Funcții/metode membre; //comportament  
} [listăDeVariabile];
```

! Nu uitați să adăugați caracterul “;” la sfârșitul declarației unei clase.

Clase. Exemplu

```
class Curs {  
    char * cursId;  
    char *nume;  
    int nrCredite;
```

*Date membre
Câmpuri
Proprietăți*

```
    void modificareNumarCredite (int);  
    void afisareIstoricCurs();  
    void informatiiEvaluariStudenti()  
    void afisareInformatii();  
};
```

Metode/Funcții membre

```
class Stiva {  
    int nrMaximDeEl;  
    int varf;  
    double * tablouEl;  
  
    double pop();  
    void push(int);  
    bool isEmpty();  
    bool isFull();  
};
```

Specificatori de acces

❑ Vizibilitatea datelor și funcțiilor membre

❑ `public`

❑ Accesate de **funcțiile membre ale clasei** și toate **funcțiile nemembre ale programului**

❑ `private`

❑ Accesate doar de **funcțiile membre** sau **prietene clasei**

❑ `protected`

❑ Asemănător cu privat, dar permite **accesul claselor derivate** la datele și funcțiile membre

❑ `default`

❑ Toate datele definite într-o clasă sunt private dacă nu este specificat un specificator de acces

Specificatori de acces

```
class Curs {  
    //Vizibilitate - Default (privată)  
    char * cursId;  
  
    //Vizibilitate - publică  
    public:  
        char *nume;  
  
    //Vizibilitate - protejată  
    protected:  
        int nrCredite;  
  
    // Vizibilitate - publica  
    public:  
        void modificareNumarCredite (int);  
        void afisareIstoricCurs();  
        void informatiiEvaluariStudenti()  
  
    //Vizibilitate - privată  
    private:  
        void afisareInformatii();  
};
```

Specificatori de acces

❑ Observații

- ❑ Domeniul de utilizare al unui specificator de acces ține până la întâlnirea altui specificator de acces
- ❑ Specificatorul de acces **implicit** este **private**
- ❑ Funcțiile membre clasei ar trebui să fie declarate publice
 - ❑ Cu excepția funcțiilor membre clasei care sunt accesate doar de alte funcții membre ale aceleiași clase

Specificatori de acces

❑ Observații

- ❑ Datele membru ale unei clase ar trebuie să fie declarate ca fiind private (private) pentru a respecta principiul încapsulării

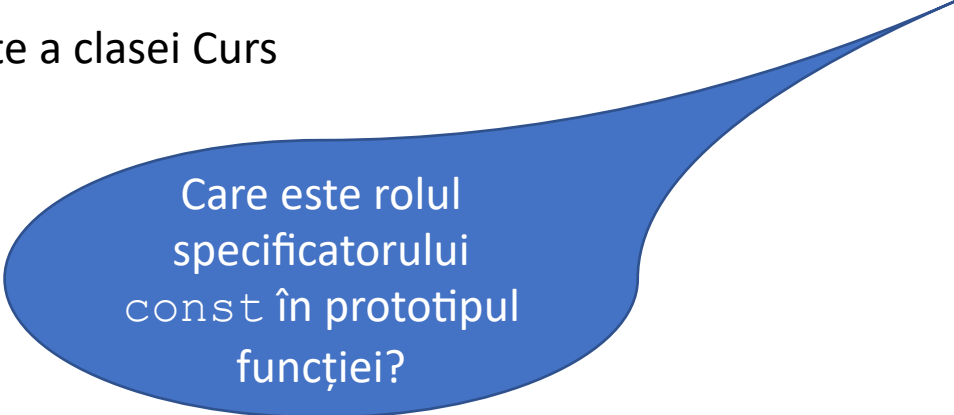
- ❑ Este util să existe funcții set și get pentru a accesa datele membru într-un mod controlat

- ❑ Exemplu funcție set pentru variabila numărCredite a clasei Curs

```
void setNumarCredite(int nrCredite) {  
    numarCredite = nrCredite;  
}
```

- ❑ Exemplu funcție get pentru variabila numărCredite a clasei Curs

```
int getNumarCredite() const {  
    return numarCredite;  
}
```



Care este rolul
specificatorului
const în prototipul
funcției?

Obiecte

❑ Instanțierea obiectelor

- ❑ Crearea unui obiect de un anumit tip

❑ Sintaxă

- ❑ `numeClasa numeObiect;`

❑ Exemplu

- ❑ `Curs p2;`
- ❑ `Curs algo;`
- ❑ `Curs comunicare;`

Pointeri la obiecte

❑ Declararea unui pointer de tip Curs

- ❑ `Curs *engleza;`

❑ Declararea și inițializarea unui pointer de tip Curs folosind constructorul implicit

❑ Varianta 1

- ❑ `Curs *engleza = new Curs(); // sau Curs *engleza = new Curs;`

❑ Varianta 2

- ❑ `Curs *engleza;`

- ❑ `engleza = new Curs();`

Tablouri de obiecte

❑ Declararea si inițializarea unui tablou de obiecte

❑ Declararea

```
❑ Curs *sir;
```

❑ Inițializarea

```
❑ sir = new Curs [10];
```

❑ Declararea unui tablou de pointeri la obiecte

❑ Declararea

```
❑ Curs **sir;
```

❑ Inițializarea

```
❑ sir = new Curs* [10];
```

❑ Inițializarea obiectelor

```
❑ sir[1] = new Curs(); sir[2] = new Curs(); ....
```

Acessarea membrilor în interiorul clasei

```
class Curs {
    char * cursId;
public:
    char *nume;
protected:
    int nrCredite;
public:
    void modificareNumarCredite (int a){
        nrCredite = a;
    }
    void afisareIstoricCurs(){
        cout << "Curs: " << nume << endl;
        informatiiEvaluariStudenti();
    }
    void informatiiEvaluariStudenti();
private:
    void afisareInformatii();
};
```

☐ Variabile membre

- ☐ pot fi accesate de toate metodele clasei indiferent de modificatori de acces

☐ Metodele unei clase

- ☐ pot apela metode definite în clasă indiferent de modificatorii de acces
- ☐ au acces la variabile membre clasei
- ☐ pot apela metode externe clasei
- ☐ au acces la variabile globale

Acesarea membrilor în exteriorul clasei

```
class Curs {  
    char * cursId;  
public:  
    char *nume;  
protected:  
    int nrCredite;  
public:  
    void modificareNumarCredite (int a);  
    void afisareIstoricCurs;  
    void informatiiEvaluariStudenti();  
private:  
    void afisareInformatii();  
};
```

☐ Variabile membre

- ☐ pot fi accesate doar dacă sunt declarate ca fiind publice

☐ Metodele unei clase

- ☐ Pot fi apelate în exteriorul clasei doar dacă sunt declarate publice

☐ Pot fi accesate cu ajutorul operatorilor

- ☐ .
- ☐ ->

prin intermediul obiectelor clasei

Acessarea membrilor în exteriorul clasei

```
class Curs {
    char * cursId;
public:
    int nrCredite;
protected:
    char *nume;
public:
    void modificareNumarCredite (int a);
    void afisareIstoricCurs;
    void informatiiEvaluariStudenti();
private:
    void afisareInformatii();
};
```

❑ Pot fi accesate cu ajutorul operatorilor

- ❑ .
- ❑ ->

prin intermediul obiectelor clasei

❑ Exemplu

```
int main() {
    Curs alg, *p2;
    Curs tab[10];
    alg.nrCredite = 8;
    p2->nrCredite = 8;
    tab[1].nrCredite = 8;
    alg.modificareNumarCredite (4);
    p2->modificareNumarCredite (4);
    tab[1].modificareNumarCredite (4);
}
```

Clase

☐ Metode speciale

☐ Constructori

- ☐ Crearea obiectelor

☐ Destructori

- ☐ Distrugerea obiectelor

Constructori

☐ Constructor

- ☐ Funcție utilizată pentru a inițializa instanțele (obiectele) unei clase

☐ Caracteristici

- ☐ Funcția are același nume cu clasa
- ☐ Nu are tip de return (nici măcar void)
- ☐ Nu poate fi funcție virtuală

☐ O clasă poate avea unul sau mai mulți constructori

- ☐ Cu număr diferit de parametri
- ☐ Constructorul implicit (default) nu are parametri

Constructori

- ❑ Procesul de creare al unui obiect

 - ❑ **Alocarea** memoriei

 - ❑ Găsirea **constructorului** corespunzător

 - ❑ Apelarea constructorului pentru a inițializa starea obiectului, membri clasei au fost anterior construiți/inițializați

Constructori. Exemplu

```
class Curs {
    char * cursId;
    char * nume;
    int nrCredite;
public:
    Curs(char *nume);
    Curs(char *nume, char *ID_curs);
    Curs(char *nume, char *ID_curs, int numarCredite);
};

void foo() {
    Curs c1 = Curs("algoritmica"); //Corect
    Curs c2("Programare III", "I2S1_P3"); //Corect
    Curs c3("Logic", "E3S1_LOG", 10); //Corect
    Curs c4; //EROARE
    Curs *c5 = new Curs("Structuri de date", "I2S2_SD"); //Corect
}
```

*Daca un constructor a fost definit
in clasa constructorul implicit nu
se mai genereaza automat*

Cum putem
simplifica, astfel
încât să folosim un
singur constructor?

Constructor. Exemplu

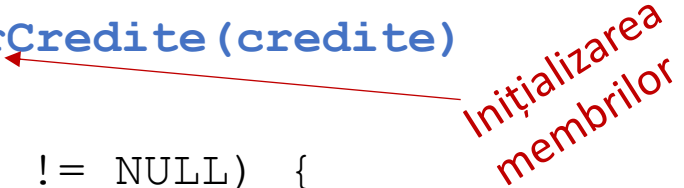
```
class Curs {  
    char * cursId;  
    char *nume;  
    int nrCredite;  
public:  
    Curs(char *nume = NULL,  
          char *ID_curs = NULL,  
          int numarCredite = 1);  
};
```

```
Curs::Curs(char *_nume, char *idCurs, int  
credite) {  
    if (_nume != NULL) {  
        nume = new char [ strlen(_nume) + 1];  
        strcpy (nume, _nume);  
    }  
    if (idCurs != NULL) {  
        cursId= new char [ strlen(idCurs) + 1];  
        strcpy( cursId, idCurs);  
    }  
    nrCredite = credite;  
}
```

Constructor. Exemplu

```
class Curs {  
    char * cursId;  
    char *nume;  
    int nrCredite;  
public:  
    Curs(char *nume = NULL,  
          char *ID_curs = NULL,  
          int numarCredite = 1);  
};
```

```
Curs::Curs(char *_nume, char *idCurs, int  
credite) : nrCredite(credite)  
{  
    if (_nume != NULL) {  
        nume = new char [ strlen(_nume) + 1];  
        strcpy (nume, _nume);  
    }  
    if (idCurs != NULL) {  
        cursId= new char [ strlen(idCurs) + 1];  
        strcpy( cursId, idCurs);  
    }  
}
```



Inițializarea
membrilor

Constructorii. Constructor implicit

- ❑ Prototip: `X ()`
- ❑ Dacă **nu există nici un constructor** definit într-o clasă, **compilatorul generează un constructor implicit**, care realizează inițializări implicite pentru datele membru ale clasei
- ❑ Dacă o clasă are membri **const sau referință** atunci constructorul implicit **nu mai este generat automat**, deoarece membri const și referință trebuie inițializați
- ❑ Dacă o clasă are definit un constructor atunci constructorul implicit **nu se mai generează automat**

Constructori. Constructor implicit

```
class Data {
public:
    // constructor implicit
    Data(int zi=0, int luna=0, int an=0);
};

class String {
public:
    String(); // constructor implicit
};

class Student {
    Data ziNastere;
    String nume;
    // constructor implicit generat automat
    // care apelează constructorii claselor Data
    // și String
};
```

```
class Data {
public:
    // NU se generează constructor implicit
    Data(int day);
};

class Test {
    const int a;
    int& r;
    // NU se generează constructor implicit
};
```

Destructor

❑ Definiție

- ❑ O funcție membră a unei clase a cărei rol este de a dealoca instanțele din memorie

❑ Sintaxă

- ❑ $\sim X ()$

❑ Caracteristici

- ❑ Funcția nu are tip de return
- ❑ Funcția nu are parametri
- ❑ Funcția este prefixată cu \sim

❑ Procesul de distrugere a unui obiect

- ❑ **Apelul** funcției destructor
- ❑ Apelarea **destructorilor datelor membru**
- ❑ **Eliberarea** memoriei

Destructor

❑ O clasă poate avea doar un destructor

❑ Compilatorul generează un destructor implicit dacă nu este unul definit în clasă

```
class Curs {
    char *idcurs;
    char *nume;
    int numarCredite;
public:
    Curs(char *nume=NULL, char *ID_curs=NULL, int numarCredite=1);
    ~Curs();
};

Curs::Curs(char *n, char *id, int nr){
    ....
    cout << "S-a creat cursul " << nume << endl;
}

Curs::~~Curs() {
    cout << "S-a sters cursul " << nume << endl;
    if (cursId != NULL) delete [] cursId;
    if (nume != NULL) delete [] nume;
}
```

```
void foo() {
    Curs c("Prog. II", "I1S2_P2", 10);
    Curs c1("Algoritmica");

    Curs *cc = new Curs("Logica");
    delete cc;
}
```

Care este rezultatul
apelului funcției
foo() ?

Destructor. Observații

- ❑ Destructori se apelează în ordinea inversă a creării obiectelor
- ❑ Dacă o clasă conține membri pointeri trebuie implementate:
 - ❑ Constructor
 - ❑ Constructor de copiere
 - ❑ Destructor
 - ❑ Supraîncărcarea operatorului =

Cuvântul cheie `this`

```
Persoana::Persoana(char *n, int an) {  
    anNastere = an;  
    if (n!=NULL) {  
        nume = new char[strlen(n)+1];  
        strcpy(nume, n);  
    }  
}
```

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

Dacă definiția constructorului este următoarea:

```
Persoana::Persoana(char *nume, int  
anNaster) {  
    anNastere = anNastere;  
    if (nume!=NULL) {  
        nume = new char[strlen(nume)+1];  
        strcpy(nume, nume);  
    }  
}
```

Cum se face diferența
între atributul clasei și
parametrul metodei?

Cuvântul cheie `this`

- Cuvântul cheie `this` – autoreferință
- Este un pointer accesibil doar dintr-un **context non-static** al unei **clase**, **structuri** sau **uniuni**
- **Pointează** spre **obiectul** pentru care **funcția membru** este apelată
- Folosirea acestuia este **implicită** dar se poate utiliza și **explicit**
- Exemplu anterior se rescrie

```
Persoana::Persoana(char *nume, int anNastere) {  
    this->anNastere = anNastere;  
    if (nume!=NULL) {  
        this->nume = new char[strlen(nume)+1];  
        strcpy(this->nume, nume);  
    }  
}
```

Se mai poate scrie:
*`(*this).anNastere`*

Cuvântul cheie `this`

❑ Folosire explicită

❑ Identificarea obiectului curent

```
Persoana Persoana::copie(const Persoana &p) {  
    if (this == &p) return *this;  
    ...  
}
```

Compararea adresei obiectului curent cu
adresa obiectului p

• Rezolvarea ambiguităților

```
class Persoana {  
    int anNastere;  
    ....  
};  
Persoana::Persoana(char *nume, int anNastre) {  
    this->anNastere = anNastere;  
    ...  
}
```

Variabilă membră clasei

Variabilă parametru formal al constructorului

Constructor de copiere

- ❑ Constructor cu un **argument** care este o **referință la clasa proprie**

- ❑ Sintaxă

 - ❑ `X (const X&obj) ;`

- ❑ Este apelat

 - ❑ La **declararea obiectelor** precum `X obj = obiect_sursă`

 - ❑ La **transmiterea argumentelor** funcțiilor `foo(X)`

 - ❑ La **crearea** unui **obiect temporar** în timpul evaluării unei expresii

Constructor de copiere

- ❑ Dacă **nu este definit** pentru o clasă, compilatorul **generează automat** unul care **copiază bit-cu-bit** conținutul obiectului sursă în obiectul destinație
- ❑ Pentru a **evita copierea** unui obiect, constructorul de copiere poate fi **declarat privat** (nu este nevoie de implementare)
- ❑ Pentru o **copie „în adâncime”** a unui obiect complex, constructorul de copiere este **obligatoriu de implementat**

Constructor de copiere

❑ Exemplu

❑ Presupunem că clasa Persoana conține metoda **void** `inlocuire (const char *s1, const char *s2);` care permite modificarea unui substring al numelui

❑ Care este rezultatul rulării următoarei secvențe de cod?

```
Persoana p("Xescu Ion", 10);
```

```
Persoana q=p;
```

```
cout<< "p=";p.afisare();
```

```
cout<< "q=";q.afisare();
```

```
p.inlocuire("Ion", "Vio");
```

```
cout<< "p=";p.afisare();
```

```
cout<< "q=";q.afisare();
```

```
q.afisare();
```

Apel implicit al constructorului de copiere generat automat de compilator (clasa nu are explicit definit constructor de copiere)

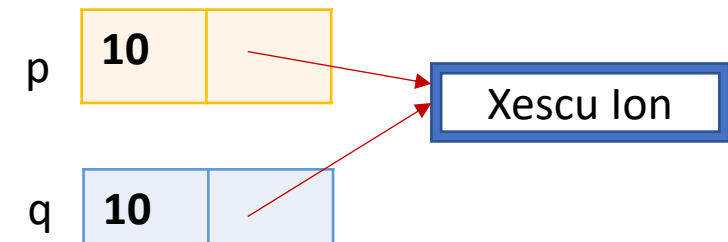
OUTPUT

```
p=(Xescu Ion, 10)
```

```
q=(Xescu Ion, 10)
```

```
p=(Xescu Vio, 10)
```

```
q=(Xescu Vio, 10)
```



Constructor de copiere. Implementare. Apel

```
class Persoana{
    char *nume;
    int anNastere;
public:
    Persoana(const Persoana &);
    ...
};

Persoana::Persoana(const Persoana &p){
    this->anNastere = p.anNastere;
    if (p.nume != NULL){
        this->nume = new char[strlen(p.nume)+1];
        strcpy(this->nume, p.nume);
    }
}

void g(Persoana d) { }

void foo() {
    // apel constructor definit de utilizator
    Persoana p("Popescu Ion", 2010);
    // apel constructor implicit
    Persoana c1;
    // apel implicit constructor de copiere
    Persoana c2 = c1;
    // apel explicit constructor de copiere
    Persoana c3(c1);
    // NU se apeleaza constructorul de copiere
    c2 = c3;
    // constructorul de copiere este apelat g(c);
}
```


Constructori. Constructor de mutare

❑ Constructor de mutare (Moving constructor)

- ❑ Introdus în standardul de C++11

- ❑ Causă

 - ❑ de multe ori se efectuau **copii inutile** a obiectelor

- ❑ Scop

 - ❑ de a împiedeca cat se poate de mult astfel de copii

- ❑ Sintaxă

 - `X (const X &&);`

- ❑ Apel

 - ❑ implicit de către compilator atunci când identifică o situație când o copie a unui obiect nu este necesară

 - ❑ explicit prin funcția `move()` din STL

Constructori. Constructor de mutare

- ❑ Este generat automat de către compilator dacă utilizatorul nu a definit explicit
 - ❑ Constructor de copiere
 - ❑ Operatorul =
 - ❑ Operatorul = de mutare
 - ❑ Destructor
- ❑ Când se apelează
 - ❑ Constructorul de copiere primește ca parametru o valoare temporară (rvalue) rezultată prin evaluare unei expresii sau returnarea unei valori a unei funcții
 - ❑ Exemple
 - ❑ `string c(a+b)`
 - ❑ `string c(fct(a))`

Constructor. Constructor de mutare. Exemplu

Constructor de copiere

```
class string{
    char * data;
    string(const string& str){
        size_t size = strlen(str.data);
        data = new char[size];
        memcpy(data, str.data, size);
    }
};
```

```
string str1(a);
```

Constructor de mutare

```
class string{
    char * data;
    string(string && str){
        data = str.data;
        str.data = nullptr;
    }
};
```

Invalidarea vechi valori pentru a evita copia în
adâncime a obiectului

```
string str2(a+b);
```

rvalue – rezultatul evaluării
expresiei $a+b$ este reținut într-o
variabilă temporară

Constructorii și excepții

❑ Cum se raportează erorile în constructor?

```
class Vector {  
public:  
    class BadSize {} ;  
    Vector(int sz)  
    {  
        if (sz < MAX_SIZE) throw BadSize();  
        // codul  
    }  
};
```

Constructorii și excepții

❑ Cum se tratează excepțiile aruncate de constructori?

```
void f(int size) {  
    try {  
        Vector v(size);  
        cout << v;  
    } catch (Vector::BadSize& bs) { }  
    cout << "Return successfully.";  
    return;  
};
```

Constructorii și excepții

❑ Cum se tratează excepțiile aruncate de constructori?

```
class Map{  
public:  
    Vector val, key;  
    Map(int sz)  
        try {  
            : val(sz), key(sz)  
        {  
        }  
    } catch (Vector::BadSize& bs) { }  
};
```

Constructori

- ❑ Tipuri

 - ❑ Impliciți

 - ❑ Definiți de utilizator

 - ❑ De copiere

 - ❑ De mutare

Curs următor

- ❑ Membri

 - ❑ statici

 - ❑ friend

- ❑ Modificatori de acces

- ❑ Relații între clase



ÎNTREBĂRI