Facultatea de Inginerie Electrică

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

EUROPEAN UNIVERSITY OF TECHNOLOGY

Facultatea de Inginerie Electrica

PCLP 2

Programarea calculatoarelor si limbaje de programare 2

PCLP2

An I semestrul II

"Coding is easy when you C it in action."

Cap. 10

Programarea orientata pe obiecte

10.1. Programarea orientata pe obiecte(POO)

- ❑ Definitie. Proprietati
- ❑ Limbaje de POO. Caracteristici
- ❑ Programarea conventionala-POO

10.2. Clase si obiecte

- ❑ Definitii. Declarare si implementare
- ❑ Specificatori de access (vizibilitate)
- ❑ Exemple
- ❑ Operatorul de rezolutie
- ❑ Accesul la componentele unei clase

❑ 10.3. Conceptul de mostenire

❑ 10.4. Constructori si destructori

❑ 10.5. Functii prietene. Supraincarcarea operatorilor

10.1 Programarea orientata pe obiecte

DEFINIRE

Programarea orientata pe obiecte (POO)= metoda de implementare in care programele sunt organizate in colectii de obiecte ce coopereaza intre ele, fiecare obiect reprezentand instanta unei clase;Fiecare clasa apartine unei ierarhii de clase, fiind conectate prin relatii de mostenire.

Limbaj POO= limbaj de programare cu suport pentru utilizarea claselor si obiectelor

Caracteristici:

- **ABSTRACTIZAREA**: abstractiune= exprima toate caracteristicile esentiale ale unui obiect, care-l fac sa se distinga de alte obiecte;
- **INCAPSULAREA**: procesul de compartimentare a elementelor. Ex.: clasa leaga intr-o singura variabila,date si functii, restrictionand accesul la acestea .
- **IERARHIZAREA:** ordonarea abstractiunilor.
- **MODULARIZAREA**: Clasele si obiectele obtinute in urma abstractizarii si incapsularii trebuie grupate si stocate intr-un modul. Modularizarea = divizarea programului in module care pot fi compilate separat, dar care sunt conectate intre ele.
- **MOSTENIREA**: defineste o relatie intre clase in care o clasa impartaseste structura si comportarea definita in una sau mai multe clase (mostenire simpla sau multipla).
- **POLIMORFISMUL**: abilitatea de a procesa obiectele diferit în functie de tipul sau de clasa lor.

10.1 Programarea orientata pe obiecte

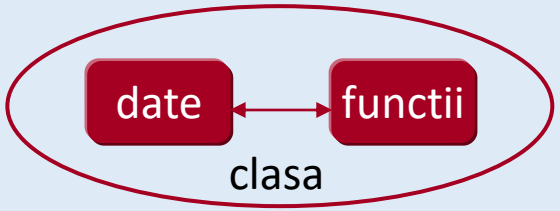
Introducere

DEFINIRE

Diferente dintre programarea conventionala (procedurala) si POO

- **programarea conventionala datele sunt preluate si retransmise de functii**
- **POO**: **clasele incapsuleaza datele si functiile intr-o singura entitate**

Programarea conventionala



Programarea orientata pe obiecte (POO)

10.2 Clase si obiecte

Introducere

DEFINIRE

Clasa = abstractizare similara cu o structura, utilizata pentru declararea unui nou tip de date care incapsuleaza date si functii;

Obiect =exemplar /instanta a unei clase

Elementele unei clase = o clasa in C++ are asociate 4 tipuri de elemente:

- ❑ colectie de **date** membre (atribute)
- ❑ colectie de **functii** membre (metode)
- ❑ **nivele de acces** ale programului (public, private, protected)
- ❑ **numele** clasei

Declararea clasei, la care se specifica:

- ❑ numele clasei,
- ❑ lista claselor de baza din care e derivata clasa, daca exista, si
- ❑ membrii clasei, atât membri de date cât si functii.

Implementarea clasei consta in:

- ❑ definitiile functiilor componente care indica comportamentul tipului de date reprezentat de clasa respectiva. Implementarea functiilor se poate face si intr-un fisier separat (.cpp).

10.2 Clase si obiecte

Introducere

SINTAXA

Sintaxa declarare clasa:

```
specificator_clasa  Nume_clasa
{ [ [ private : ] lista_membri_1]
  [ [ public : ]   lista_membri_2]
};
```

unde:

specificator_clasa poate fi:

- class;
- struct;
- union;

Obs. Diferenţa principală între specificatorii “class”, “struct” şi “union”:

- pentru o clasă declarată cu “class”, datele membre sunt implicit de tip private, până la prima folosire a unui specificatori de acces public sau protected.
- pentru o clasă declarată cu “struct” sau “union”, datele membre sunt implicit de tip public, până la prima folosire a unuia din specificatorii private sau protected

10.2 Clase si obiecte

Declararea si implementarea claselor: class

```
Format de declarare:
class   nume_clasa
            {
            variabile si functii particulare
specificator de acces:
            variabile si functii
specificator de acces:
            variabile si functii
...
specificator de acces:
            variabile si functii
            } [Lista_de_obiecte];
```

variabilele se declara cu tip:
int,double,float,char,etc.

functiile se implementeaza astfel:
-functii inline (in interiorul clasei) sau
-operatorul de rezolutie ::(inafara clasei)

specificatori de acces pot fi:
public
private
protected

Lista_de_obiecte este optionala.
Declararea obiectelor se poate face si astfel: **nume_clasa Lista_de_obiecte;**

10.2 Clase si obiecte

Specificatori de acces (vizibilitate)

Definitii

- Implicit, variabilele si functiile declarate intr-o clasa sunt proprii (private) acelei clase si numai membri ei au access la ele.
- Specificatorii de acces au efect pana cand se intalneste un alt specificator de acces sau se ajunge la sfarsitul declaratiei de clasa.
- Specificatorii de acces pot alterna in declararea unei clase. public, private,public, etc....

Tipuri specificatori de access:

- public: datele si functiile declarate cu acest specificator sunt vizibile (accesibile) din orice zona a programului.
- private: datele si functiile declarate cu acest specificator pot fi accesate doar de catre membri clasei.
- protected: datele si functiile declarate cu acest specificator pot fi modificate doar in cadrul clasei sau in clasele derivate din ea

10.2 Clase si obiecte

Exemple clase

EXEMPLE

Ex. Alternarea public-private in declararea unei clase de tip angajat

```
class  angajat {
        char nume[20];
public:
        void citestenume(char *n);
        void preianume(char *n);
private:
        double salar;
public:
        void citestesalar(double w);
        double preiasalar();
};

angajat pers;
```

Variabile (date) membre

Functii membre

Obiect pers de tipul clasa angajat

10.2 Clase si obiecte

Exemple clase

EXEMPLE

Ex. Definirea unei functii membre in interiorul clasei

```
class adunare {
public:
        int suma(int a,int b) {
        int rez;
        rez=a+b;
        return rez;}
};
//Utilizand o functie inline
```

Ex. Definirea unei functii membre in exteriorul clasei

```
class adunare {
public:
        int suma(int a,int b) ;
};
int adunare::suma(int a,int b) {
        int rez;
        rez=a+b;
        return rez;
}
//Utilizand operatorul de rezolutie
```

10.2 Clase si obiecte

Exemple clase

EXEMPLE

Ex. Calculul volumului unei cutii utilizand clase (numai cu date membre)

```
#include <iostream>
using namespace std;
class Box {
   public:
      double lungime;   // Lungime cutie
      double latime;  // Latime cutie
      double inaltime; };  // Inaltime cutie
int main() {
Box Box1        // Declara Box1 de tip Box
double volume = 1;     // declara variabila volume
Box1.inaltime = 5.0;  //initializare cutie
Box1.lungime = 6.0;
Box1.latime = 7.0;
volume = Box1.inaltime * Box1.lungime * Box1.latime;
cout << "Volumul  Box1 : " << volume <<endl;
return 0;}
```

```
Volumul  Box1 : 210
```

10.2 Clase si obiecte

Operatorul de rezolutie ::

Definitii

Operatorul de rezolutie :: = operator de specificare a domeniului, specifica din ce domeniu (clasa) face parte o anumita functie

Mai multe clase diferite pot sa folosesca acelasi nume de functie membra !

Definirea functiilor membre ale unei clase se poate face :

- în exteriorul clasei prin specificarea numelui clasei urmat de operatorul de rezolutie :: înaintea numelui functiei,
- ca functii inline, la declararea lor în interiorul clasei.

10.2 Clase si obiecte

Accesul la componentele unei clase

Definitii

Accesul la componentele unei clase se realizeaza similar cu accesul la elementele unei structuri, prin:

1. Instantierea (declararea) unui obiect

```
nume_clasa   nume_obiect;
nume_clasa   *nume_pointer;
```

2. Accesarea componentelor clasei prin
   - operatorul **.**

```
nume_obiect.nume_functie;      nume_obiect.nume_data;
```

   - operatorul **->**

```
nume_pointer->nume_functie;      nume_pointer->nume_data;
```

10.2 Clase si obiecte

Exemple

EXEMPLE

Ex.1 Calculul volumului unei cutii utilizand clase (cu date si functie membre)

```
#include <iostream>
using namespace std;
class Box {
            public:
                double lungime;   // Lungime cutie
                double latime;  // Latime cutie
                double inaltime;   // Inaltime cutie
                double volume();         };
double Box::volume()
  {return lungime*latime*inaltime;}
int main() {
   Box Box1;        // Declara Box1 de tip Box
   Box1.inaltime = 5.0;
   Box1.lungime = 6.0;
   Box1.latime = 7.0;
   cout << "Volum Box1 : " << Box1.volume() <<endl;   return 0;       }
```

```
Volum Box1 : 210
```

10.2 Clase si obiecte

Exemple

EXEMPLE

Ex. 2 Se declara o clasa in care functia membru se defineste in exteriorul clasei

```
#include <iostream>
using namespace std;
class zi_din_an  {
public:
        int zi;  //data membru
        int luna; //data membru
        void afisare();};//functie membru
int main()
{zi_din_an   azi, zi_nastere; //se declara 2 obiecte de tip clasa
cout << "Dati data de azi (zi luna, ex: 7 6):\n"; cin >> azi.zi >> azi.luna;
cout << "Dati ziua de nastere:\n"; cin >> zi_nastere.zi >> zi_nastere.luna;
cout << "Azi suntem in "; azi.afisare(); cout << "Ziua dvs de nastere este ";
zi_nastere.afisare();
if (azi.luna==zi_nastere.luna && azi.zi==zi_nastere.zi) cout << "La multi ani!\n";
else   cout << "O zi buna !\n"; return 0;}
void zi_din_an :: afisare()
                {    cout << "luna=" << luna  << ",ziua" << zi << endl;}
```

```
Dati data de azi (zi luna, ex: 7 6):
25 11
Dati ziua de nastere:
26 11
Azi suntem in luna=11,ziua25
Ziua dvs de nastere este luna=11,ziua26
O zi buna !
```

TEST

Se consideră secvenţa de instrucţiuni (2p):

```
class angajat{
                    char nume[25];
                    double salar;
public:
                    double impozit();  };
```

Indicaţi instrucţiunea corectă pentru definirea funcţiei membre impozit() din clasa student in exteriorul clasei:

a) double class::impozit(){…}
b) void class:: impozit() {…}
c) double angajat::impozit(){ …}
d) class angajat::void impozit {…}

Raspuns correct

c)

10.2 Clase si obiecte

Exemple

EXEMPLE

Ex.Calculul volumului unei cutii utilizand un obiect de tip clasa si un pointer

```
#include <iostream>
using namespace std;
class Box {
        public:
            double lungime;   // Lungime cutie
            double latime;  // Latime cutie
            double inaltime;   // Inaltime cutie
            double volume();
        };
double Box::volume()
  {return lungime*latime*inaltime;}
int main() {
   Box Box1, *p;        // Declara un obiect Box1 si un pointer *p de tip Box
   p=&Box1;
   p->inaltime = 5.0;
   p->lungime = 6.0;
   p->latime = 7.0;   cout << "Volum Box1 : " << p->volume() <<endl;
   cout << "Volum Box1 : " << (*p).volume();    return 0;    }
```

```
Volum Box1 : 210
Volum Box1 : 210
```

TEST kahoot

Pentru login, introduceti codul afisat pe ecran, in browser la adresa:

http://kahoot.it

10.3. Conceptul de mostenire

Accesul la componentele unei clase

Definitii

Mostenirea = permite construirea unei ierarhii de clase.

Procesul de ierarhizare consta in :

- creearea unei clase de baza (“parinte”=“parent”): cea mai generala descriere care stabileste calitatile comune ale tuturor obiectelor ce vor deriva din aceasta baza
- creearea claselor derivate (“copii”=“child”): din clasa de baza, care vor include :
  - toate caracteristicile clasei de baza si in plus
  - calitati propri clasei respective.

Forma generala de declarare clase derivate prin mostenire:

```
class nume_nou_clasa: [specificator_acces] clasa_mostenita
{
//corpul noii clase
}
```

unde clasa_mostenita= optional

10.3. Conceptul de mostenire

Specificatori de acces si reprezentare grafica

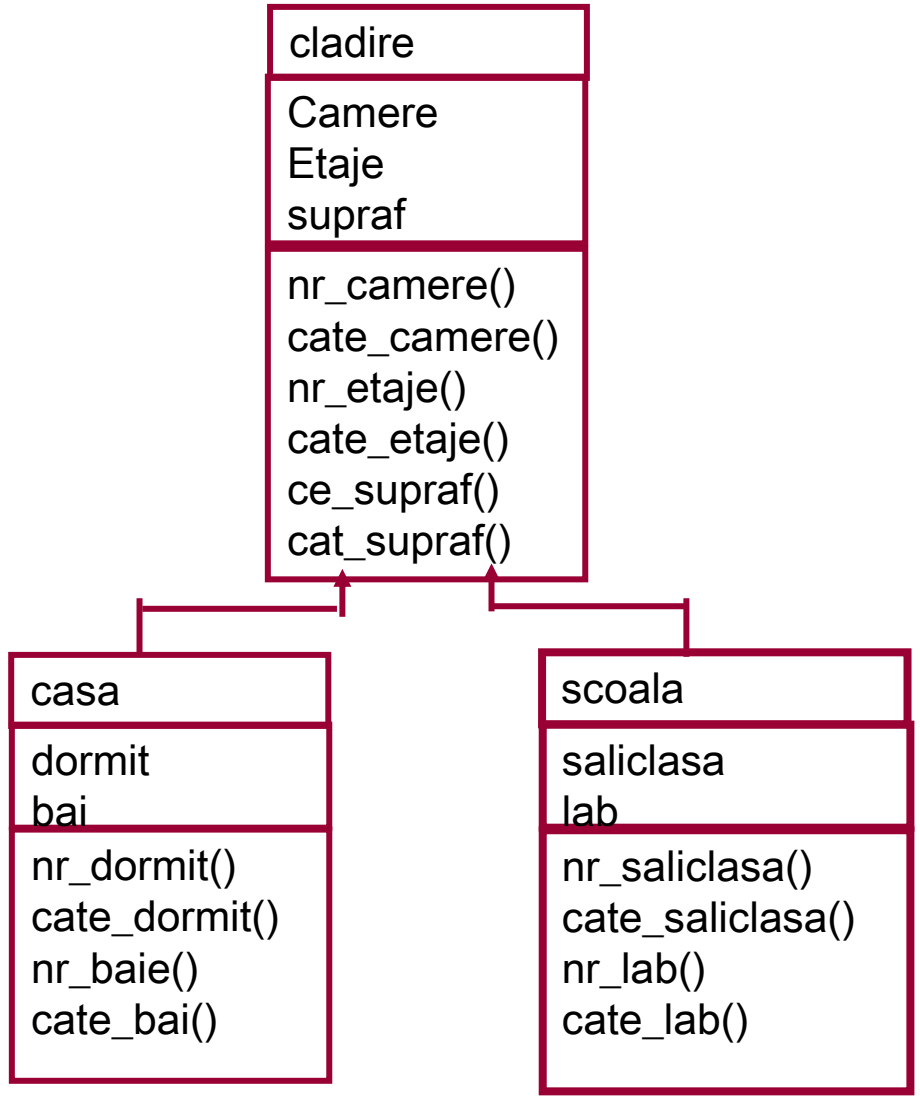
SPECIFICATOR ACCES	ACCES PERMIS PENTRU
public	Toate celelalte clase
private	Nici o alta clasa
protected	Numai claselor derivate

nume
Date (atribute)
Metode(functii)

EXEMPLE

Ex.1 ierarhizare clase

- clasa de baza: cladire,
- clase derivate: casa si scoala



10.3. Conceptul de mostenire

EXEMPLE

Ex.1. Definirea clasei de baza:

```
class cladire {
        int camere;
        int etaje;
        int supraf;
public:
        void nr_camere(int num);
        int cate_camere();
        void nr_etaje(int num);
        int cate_etaje();
        void ce_supraf(int num);
        int cat_supraf();
};
```

Definirea unei clasei derivate

```
class casa: public cladire {
        int dormit;
        int bai;
public:
        void nr_dormit(int num);
        int cate_dormit();
        void nr_bai(int num);
        int cate_bai();};
```

Definirea altei clasei derivate

```
class scoala: public cladire {
        int saliclasa;
        int lab;
public:
        void nr_saliclasa(int num);
        int cate_saliclasa();
        void nr_lab(int num);
        int cate_lab();};
```

10.3. Conceptul de mostenire

EXEMPLE

Definirea functiilor din clasa de baza: cladire

```
void cladire::nr_camere(int num)
{ camere=num;
}
void cladire::nr_etaje(int num)
{etaje=num;
}
void cladire::ce_supraf(int num)
{supraf=num;
}
int cladire::cate_camere()
{return camere;
}
int cladire::cate_etaje()
{return etaje ;
}
int cladire::cat_supraf()
{return supraf ;
}
```

Definirea functiilor din clasa derivata: casa

```
void casa::nr_dormit (int num)
{
dormit=num;
}
void casa::nr_bai (int num)
{
bai=num;
}
int casa::cate_dormit ()
{
return dormit;
}
int casa::cate_bai ()
{
return bai;
}
```

Definirea functiilor din clasa derivata: scoala

```
void scoala::nr_saliclasa(int num)
{
saliclasa=num;
}
void scoala::nr_lab(int num)
{
lab=num;
}
int scoala::cate_saliclasa()
{
return saliclasa;
}
int scoala::cate_lab()
{
return lab;
}
```

10.3. Conceptul de mostenire

EXEMPLE

Mod de implementare:

- ❑se declara clasa de baza
- ❑se declara cele 2 clase derivate
- ❑se declara 2 obiecte de tipul claselor derivate casa: c si respectiv scoala: s
- ❑se definesc functiile membre din toate cele 3 clase
- ❑se defineste functia main in care:
  - se atribuie valori functiilor membre ale obiectelor declarate
  - se afiseaza pe ecran diferite caracteristici ale celor 2 obiecte

10.3. Conceptul de mostenire

EXEMPLE

```
int main()
{ casa c; //obiect de tip casa
scoala s; //obiect de tip scoala
c.nr_camere(12);     c.nr_etaje(3);
c.ce_supraf(4500);  c.nr_dormit(5); c.nr_bai(3);
cout << "Casa are suprafata:" <<c.cat_supraf() <<" mp," ;
cout << c.cate_etaje() << " etaje, si "       ;
cout << c.cate_camere() << " camere, din care:\n";
cout << "- dormitoare: " << c.cate_dormit() <<endl;
cout << "- camere diferite de dormitoare: " << c.cate_camere(); cout <<- c.cate_dormit() <<endl;
cout << "- bai: " << c.cate_bai() <<endl;
s.nr_camere(200);    s.nr_etaje(3); s.nr_saliclasa(180);  s.nr_lab(5); s.ce_supraf(25000);
cout << "\nScoala are suprafata: " << s.cat_supraf() <<" mp," ;
cout << s.cate_etaje() << "etaje, si " <<s.cate_camere() << " ;
cout << incaperi, din care:\n"; cout << "-sali de clasa: " << s.cate_saliclasa() << endl;
cout << "-laboratoare: " << s.cate_lab() <<endl;
cout << "-alte sali (magazii, etc) : " ;
cout << s.cate_camere() - s.cate_saliclasa()- s.cate_lab() ; return 0;}
```

```
Casa are suprafata:4500 mp,3 etaje, si 12 camere, din care:
-dormitoare:5
-camere diferite de dormitoare:7
-bai:3
Scoala are suprafata: 25000 mp,3etaje, si 200 incaperi, din care:
-Sali de clasa: 180
-laboratoare: 5
-alte sali (magazii, etc):15
```

10.3. Conceptul de mostenire

Aplicatii cu clase

EXEMPLE

Ex.2 Implementare clasa student si calcul medii

```
#include<iostream>
#define N 25
using namespace std;
class student{
                char nume[25];
                char prenume[30];
                int nota[5];
public:
                void id_student();
                void afis();
                double media();};
int n;
void student::id_student(){
   cout<<"\nNume student:"; cin>>nume;
   cout<<"Prenume student:";cin>>prenume;
   for(int k=1;k<=5;k++)
      {cout<<"Nota"<<k<<":";  cin>>nota[k];}
}
```

```
void student::afis(){
   cout<<"\nNume student:“;cout<<nume;
   cout<<"\nPrenume student:“;cout<<prenume<<"\n";
   for(int k=1;k<=5;k++)
      {cout<<"Nota:"<<k<< ":" <<nota[k] <<endl;}
   cout<<"Media:"<<media()<< endl;}
double student::media(){
   double m=.0;
   for(int k=1;k<=5;k++)
        if(nota[k]<5){
          cout<<"\n Student restantier";
          return 0;}
        else m+=nota[k];   return m/5;}
int main(){
student s[N];
cout<<"\n Numarul de studenti:"; cin>>n;
for(int i=1;i<=n;i++)
   {s[i].id_student(); s[i].afis();}
return 0;}
```

```
Numarul de studenti:2

Nume student:Pop
Prenume student:Lucia
Nota1:10
Nota2:9
Nota3:8
Nota4:9
Nota5:10

Nume student:Pop
Prenume student:Lucia
Nota:1:10
Nota:2:9
Nota:3:8
Nota:4:9
Nota:5:10
Media:9.2

Nume student:Avram
Prenume student:David
Nota1:5
Nota2:4
Nota3:6
Nota4:7
Nota5:7

Nume student:Avram
Prenume student:David
Nota:1:5
Nota:2:4
Nota:3:6
Nota:4:7
Nota:5:7

Student restantierMedia:0
```

10.3. Conceptul de mostenire

Aplicatii cu clase

EXEMPLE

Ex.3 Implementare clasa si calculul ariei dreptunghi (fara mostenire)

```
#include <iostream>
using namespace std;
class drept {
    int latime, lungime;
  public:
    void set_values (int,int);
    int aria() {return latime*lungime;} // functie inline
};
void drept::set_values (int x, int y)  //functie explicitata in exteriorul clasei
{  latime = x;
   lungime= y;
}
int main () {
  drept pa; //instantiere obiect de tip clasa dreptunghi
  pa.set_values (3,4);
  cout << "area: " << pa.aria();
  return 0; }
```

```
area: 12
```

10.4. Constructori si destructori

Definire constructori /destructori

Definitii

Functie constructor = functie speciala care este membru al unei clase si are acelasi nume cu acea clasa, nu poate sa returneze valori si nu contine nici tipul de returnat

Un **destructor al unui obiect constructor al unui obiect** este apelat atunci cand este creat acel obiect. Un va produce distrugerea obiectului creat (pentru eliberarea de memorie, etc.)

Ordinea de apelare a constructorilor si destructorilor

- Constructori: clasa baza ⇒clasa derivata1 ⇒clasa derivata2
- Destructori: clasa derivata2 ⇒clasa derivata1 ⇒clasa baza

Initializarea obiectelor

Inițializarea obiectelor

IdNumeClasa idObiect(<listaParametri>); sau

IdNumeClasa idObiect = valParam daca lista de parametri e formată dintr-un singur parametru

Inițializarea la declarare a obiectelor se poate face prin intermediul unor funcții speciale numite **constructori**. Constructorii pot fi de mai multe tipuri: impliciti, de copiere, etc.

10.4. Constructori si destructori

Definire constructori /destructori

EXEMPLE

Ex: Initializare obiecte prin functii membre:

```
class Dreptunghi {
    int latime, inaltime;
  public:
    void seteaza_valori (int,int);
    int aria() {return latime*inaltime;}//inline
};
void Dreptunghi::seteaza_valori (int x, int y) {
  latime = x;
  inaltime = y;}

int main () {
  Dreptunghi drept;
  drept.seteaza_valori (3,4);
  cout << "aria: " << drept.aria();  return 0;}
```

```
area: 12
```

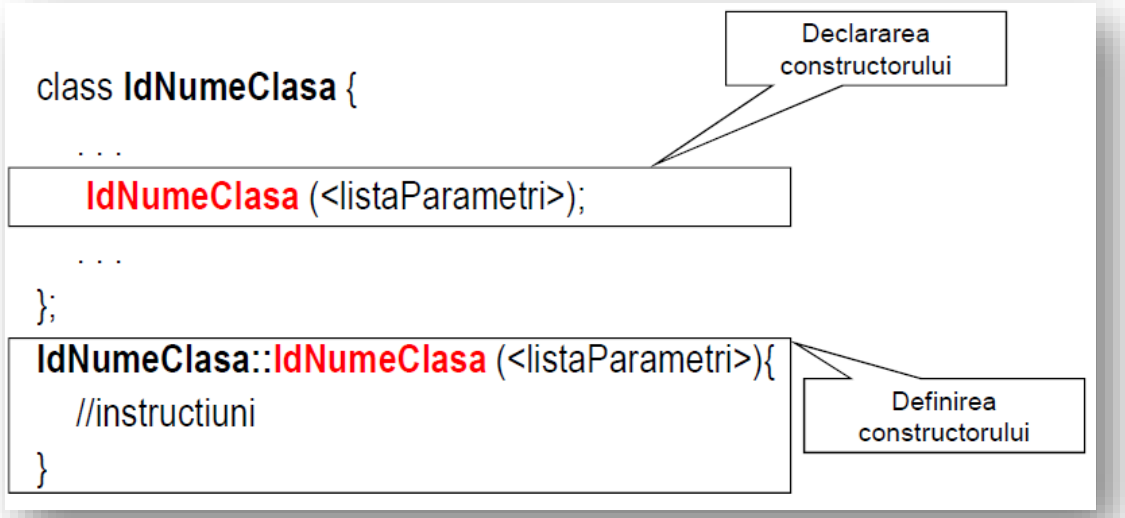
10.4. Constructori si destructori

Constructori impliciti

Definitii

Caracteristici

- au același nume cu cel al clasei din care fac parte
- nu returnează nimic (nici măcar tipul **void**)
- o clasă poate avea mai mulți constructori
- nu pot primi ca parametri instanțe ale clasei ce se definește, ci doar pointeri sau referințe la instanțele clasei respective
- constructorii nu sunt apelaţi explicit (în general)
- constructorii nu se *moștenesc*



10.4. Constructori si destructori

Initializare obiecte prin constructor

EXEMPLE

Ex: Initializare prin constructor

```
#include <iostream>
using namespace std;
class Dreptunghi {
   int latime, inaltime;
  public:
   Dreptunghi (int,int);
   int aria () {return (latime*inaltime);}};
Dreptunghi::Dreptunghi (int a, int b) {
  latime = a;
  inaltime = b;}
int main () {
  Dreptunghi drept (3,4);
  Dreptunghi dreptb (5,6);
  cout << "aria lui drept: " << drept.aria() << endl;
  cout << "aria lui dreptb: " << dreptb.aria() << endl;  return 0;}
```

```
aria lui drept: 12
aria lui dreptb: 30
```

10.4. Constructori si destructori

Initializare obiecte prin constructor

EXEMPLE

Ex: fie clasa Complex, initializarea membrilor nu se poate face in main() direct pentru ca aceste date sunt private. Daca ar fi public codul de initializare ar fi corect:

```
class Complex {
private:
        float re;   float im;
public:
        void citire();
        void afisare();
        float modul();
};
int main(){
Complex z;
z.re = 2.5; // Incorect deoarece re ,im sunt private
z.im=5;
Complex *p=&z;
p -> afisare();     return 0;}
```

10.4. Constructori si destructori

Constructori impliciti

EXEMPLE

Ex: Mod de initializare : cu constructor implicit , var. 1

```
class Complex {
private:
    float re;  float im;
public:
    Complex(float r, float i);
    void citire();
    void afisare();
    float modul();    };
void Complex::citire(){
    cout <<"partea reala:";cin>>re;
    cout <<"partea imaginara:"; cin>>im;}
void Complex::afisare(){cout<<re<<" "<<im;}
float Complex::modul(){return sqrt(re*re + im*im);}
Complex::Complex(float r, float i){ //constructor
re = r; im = i;}
```

```
int main(){
Complex z(7,3);
z.afisare();
return 0;}
```

```
7 3
```

10.4. Constructori si destructori

Constructori impliciti

EXEMPLE

Mod de initializare : cu constructor implicit varianta 2

```
class Complex {
private:
    float re;     float im;
public:
    Complex(){
    re = 0;     im = 0;
cout <<"Apel constructor\n";}
   void citire();
    void afisare();
    float modul();     };
void Complex::citire(){
cout <<"partea reala:";cin>>re;
cout <<"partea imaginara:"; cin>>im; }
void Complex::afisare(){
cout<<re<<" "<<im;}
float Complex::modul(){
return sqrt(re*re + im*im);}
```

```
int main(){
Complex z;
z.afisare();
    return 0;}
```

```
Apel constructor
0 0
```

10.4. Constructori si destructori

Constructori impliciti

EXEMPLE

Mod de initializare : constructor cu parametri impliciți varianta 3

```
class Complex {
private:
    float re;     float im;
public:
    Complex(float r=0,float i=0){
      re = r;
      im = i;}
    void citire();
    void afisare();
    float modul();     };
void Complex::citire(){
  cout <<"partea reala:";cin>>re;
  cout <<"partea imaginara:"; cin>>im; }
void Complex::afisare(){
  cout<<re<<" "<<im;}
float Complex::modul(){
  return sqrt(re*re + im*im);}
```

```
int main(){
Complex z1(2,3),  z2(4),  z3=5,  z4;
z1.afisare();cout<<" ,";
z2.afisare();cout<<" ,";
z3.afisare();cout<<" ,";
z4.afisare();
return 0;}
```

```
2 3 ,4 0 ,5 0 ,0 0
```

10.4. Constructori si destructori

Constructori de copiere

Definitii

Constructori de copiere – inițializarea obiectelor la declarare cu alte obiecte deja create

- definiți de utilizator
- generați de compilator

```
class IdNumeClasa {
   . . .
    IdNumeClasa (IdNumeClasa &ob);
        sau
    IdNumeClasa (const IdNumeClasa &ob);
   . . .
};
```

Recomandat

10.4. Constructori si destructori

Constructori de copiere

EXEMPLE

Mod de initializare : constructor cu parametri impliciți varianta 4

```
class Complex {
private:
    float re;
    float im;
public:
    Complex(float real = 0, float imag = 0) : re(real), im(imag) {cout << "Apel constructor\n";    }
    Complex(const Complex &z) { re = z.re; im = z.im; cout << "Apel constructor de copiere\n";    }
    void citire() {
        cout << "Introduceti partea reala: "; cin >> re;
        cout << "Introduceti partea imaginara: "; cin >> im;    }
    void afisare() {
        cout << "Partea reala: " << re << ", Partea imaginara: " << im << endl;   }
    float modul() { return sqrt(re * re + im * im);    }};
int main() {
Complex z1(2, 3);    z1.afisare();
Complex z2 = z1;     z2.afisare();
return 0;}
```

```
Apel constructor
Partea reala: 2, Partea imaginara: 3
Apel constructor de copiere
Partea reala: 2, Partea imaginara: 3
```

10.4. Constructori si destructori

Destructori

Definitii

Destructorul este o funcție membră speciala a unei clase ce apelează în mod automat distrugerea unui obiect

EXEMPLE

```
class Complex {
private:
    float re;     float im;
public:
    Complex(float r, float i);
    ~Complex();
    void citire();
    void afisare();
    float modul();    };
…
Complex::Complex(float r, float i){
   re = r; im = i;}
Complex::~Complex(){
   cout<<" Distrugere obiect:";
   afisare();}
```

```
int main(){
Complex z1(2,3), z2(4,7);
z1.afisare();
z2.afisare();
return 0; }
```

```
2 3,4 7, Distrugere obiect:4 7, Distrugere obiect:2 3,
```

10.5. Functii prietene

SINTAXA

Functie prietena (friend) = functie care are acces la membri private si protected ai clasei careia ii este prietena.

Format de declarare:

```
class nume_clasa {
    variabile si functii private
public:
    friend prototip_fctie();
    variabile si functii publice;
};
```

Functiile prietene sunt utile la:

- ❑ supraincarcarea operatorilor
- ❑ simplifica implementarii functiilor de I/O
- ❑ scrierea mai eficienta a programelor

10.5. Functii prietene

EXEMPLE

Ex. Sa se scrie un program in C++ care defineste o clasa exemplu cu 2 date membre de tip intreg si o functie prietena (friend) care realizeaza suma a doua numere intregi

```
#include  <iostream>
using namespace std;
class exemplu {
     int  a, b;
public:
     friend int sum(exemplu x);
     void set_ab(int i, int j);};
void exemplu::set_ab (int i, int j)
{    a = i;    b = j;}
// Obs: sum() nu este functie membra a clasei.
// sum() este functie prietena a clasei "exemplu"si poate accesa direct pe a si b
int sum (exemplu x)
{return x.a + x.b;}
int main()
{     exemplu n; n.set_ab(10, 20);
      cout << "Suma :10+20=" << sum(n) << "\n"; return 0;}
```

```
Suma :10+20=30
```

10.5. Functii prietene

Supraincarcarea operatorilor

EXEMPLE

Ex. Functiile membre print() au efect diferit desi au acelasi nume .

```
#include <iostream>
using namespace std;
class printData {
public:
        void print(int i) { cout << "Print int: " << i << endl; }
        void print(double f) { cout << "Print double: " << f << endl; }
        void print(char* c) { cout << "Print caractere: " << c << endl; }
};
int main(void)
{ printData pd;
pd.print(5); // apel print integer
pd.print(500.263); // apel print double
pd.print("Hello C++"); // apel print caracter
return 0; }
```

```
Print int: 5
Print double: 500.263
Print caractere: Hello C++
```