Cursul 3

- De la C la C++, elemente de programare structurată(continuare)
  - Supraîncarcarea operatorilor
  - Template – uri
  - Alocare dinamică

Supraîncarcare operatori 1

- O nouă modalitate de a apela o funcție
- Se pot supraîncarca operatorii existenți, cu excepția operatorilor:
  - : : Operatorul de rezoluție
  - . Operatorul de selectare a membrilor
  - .* Operatorul de selectare prin pointeri la funcții
- Nu pot fi introduşi operatori noi
- Nu pot fi schimbate: precedența şi asociativitatea

Supraîncarcare operatori 2

- Sintaxa este cea de la funcții; numele funcției este:

  `operator@` (@ este operatorul ce se supraîncarcă)

- Numărul argumentelor depinde de:
  - Aritatea operatorului (unar sau binar)
  - Locul defininirii:
    - Ca funcție globală: 1 pentru unar, 2 pentru binar;
    - Ca funcție membru : 0 pentru unar, 1 pentru binar

Supraîncarcare operatori 3

- Apelul:

```
a @ b                    @a    a@
a.operator@(b)    a.operator@()
```

- Excepții ++ şi --
  - prefixați: se respectă regula
  - postfixați: un argument fictiv

```
operator++();// ++ prefixat
operator++(int);// ++ postfixat
```

Exemplul 1

```
struct Clock{
    Clock tic_tac();
    Clock operator++();
    Clock operator++(int);
    int ora, min, ap; // ap=0 pentru AM, 1 pentru PM
};
Clock Clock::tic_tac(){
    ++min;
    if(min == 60){
        ora++;
        min = 0;
    }
    if(ora == 13)
        ora = 1;
    if(ora == 12 && min == 0)
        ap = !ap;
    return *this;
}
```

Exemplul 1

```
Clock Clock::operator++(){
   return tic_tac();
}
Clock Clock::operator++(int n){
   Clock c = *this;
   tic_tac();
   return c;
}
```

Exemplul 1

```
ostream& operator<<(ostream& out, Clock c){
    out << setfill ('0') << setw(2) << c.ora
        << ':' << setw(2) << c.min << setfill(' ');
    if(c.ap)
        out << " PM";
    else
        out << " AM";
    return out;
}
int main(){
    Clock x , y;
    x.ora = 11;x.min = 59; x.ap = 0;
    cout << x.ora << x.min << x.ap << endl;
    y = x++;
    cout << " x = " << x << endl;
    cout << " y = " << y << endl;
    x.tic_tac(); x.tic_tac();
    cout << " x = " << x << endl;
    return 0;
}
```

Exemplul 2

```
using namespace std;
struct complex{
    float re;
    float im;
};
int operator==(complex, complex);
int operator!=(complex, complex);
complex operator+(complex, complex);
complex operator+(int, complex);
complex& operator+=(complex&, complex);
complex operator-(complex, complex);
complex& operator-=(complex&, complex);
complex operator*(complex, complex);
complex& operator*=(complex&, complex);
complex operator/(complex, complex);
complex& operator/=(complex&, complex);

ostream& operator<<(ostream&, complex&);
istream& operator>>(istream&, complex&);
```

Exemplul 2

```
complex operator+(complex z1, complex z2){
            z1.re += z2.re;
            z1.im += z2.im;
            return z1;
}
complex operator+(int k, complex z){
            z.re += k;
            z.im += k;
            return z;
}


complex& operator+=(complex& z1, complex z2){
   z1.re += z2.re;
   z1.im += z2.im;
   return z1;
}
```

Exemplul 2

```
int operator!=(complex z1, complex z2){
   return (z1.re != z2.re) || (z1.im != z2.im);
}
```

```
ostream& operator<<(ostream& s, complex& z){
   s << "(" << z.re << ", " << z.im << ")";
   return s;
}
```

```
istream& operator>>(istream& s, complex& z){
   s >> z.re >> z.im;
   return s;
}
```

Exemplul 3

```
struct Punct{
    int x, y;
};
Punct operator*(int n, Punct p){ // Numar * Punct
    Punct q;
    q.x = n*p.x;
    q.y = n*p.y;
    return q;
}
bool operator==(Punct p, Punct q){ // p1==p2?
    return (p.x == q.x && p.y == q.y);
}

ostream& operator<<(ostream& s, Punct& p){
   s << "(" << p.x << ", " << p.y << ")";
   return s;
}
```

Exemplul 3

```
istream& operator>>(istream& s, Punct& p){
   s >> p.x >> p.y;
   return s;
}


Punct p;
int n = 3;
cin >> q;
p = n*q;
cout << p << q;
if(p == q) cout << "p egal q";
else cout << "p diferit q";
```

Operatori de conversie

```
class X{
public:
    X(double = 0.0l);
    ~X();
    operator int();
private:
    double dbl;
};
//...
X::operator int(){
return int(dbl);
}
X x(3.14);
int i = int(x);
int j = (int) x;
int k = x;
```

Operatori de conversie

```
#include <iostream.h>
#include <assert.h>
class Int{
public:
     Int(int a = 0) : i(a) {}
     operator int() const{return i;}
private:
     int i;
};
void main(){
     Int i;
     i = 12;
     int ibis = i; // apel operator int()
     if (ibis == i) // apel operator int()
        cout << "Un intreg Int: " << i << endl; //apel operator int()
     Int j = 3, k = j, m(k);
     assert(m + .1416 == 3.1416); // apel operator int()
     cout << "Sfarsit de program!"<< endl;
}
/*
Un intreg Int: 12
Sfarsit de program!
*/
```

Template-uri

```
//Supraincarcare functii
void swap(int& x, int& y){
  int aux = x;
  x = y;
  y = aux;
}
void swap(char& x, char& y){
  char aux = x;
  x = y;
  y = aux;
}
void swap(float& x, float& y){
  float aux = x;
  x = y;
  y = aux;
}
```

Template-uri

```
int x = 23, y = 14;
char c1 = 'o', c2 = '9';
double u = .5, v = 5.5;
swap(x, y);
swap(u, v);
swap(c1, c2);
```

- Polimorfism parametric:
  - utilizarea aceluiaşi nume pentru mai multe înțelesuri(polimorfism).
  - înțelesul este dedus din tipul parametrilor (parametric).

Template-uri

- Funcție generică de interschimbare:

```
template <class tip>
void swap(tip& x, tip& y) {
  tip aux = x;
  x = y;
  y = aux;
}

int m = 8, n = 15;
swap(m, n); // swap<int>(m, n);
cout << endl << m << ", " << n << endl;
double a = 10.0, b = 20.0;
swap(a,b); // swap<double>(m, n);
cout << endl << a << ", " << b << endl;
complex z1, z2;
z1.re = 23.5; z1.im = 33.5;
z2.re = 1; z2.im = 2;
swap(z1, z2); // swap<complex>(m, n);
```

Template-uri

- Funcție generică de sortare:

```
template <class T>
void naiveSort(T a[], int n)
{
    for (int i=0; i<n-1; i++)
        for(int j=i+1; j<n; j++)
            if (a[i] > a[j])
                swap<T>(a[i], a[j]);
};
```

- Parametrii generici pot fi şi tipuri de bază

```
template <class T, int n>
void naivSort2(T a[])
{
    for (int i=0; i<n-1; i++)
        for(int j=i+1; j<n; j++)
            if (a[i] > a[j])
                swap<T>(a[i], a[j]);
};
```

Template-uri

- La apel, pentru n trebuie transmisă o constantă:

```
int i;
double x[5] = {2,1,3,5,4};
naivSort<double>(x, 5);
for (i=0; i<5; i++)
  cout << x[i] << ", ";
cout << endl;
double y[5] = {2,1,3,5,4};
naivSort2<double, 5>(y);
for (i=0; i<5; i++)
  cout << x[i] << ", ";
cout << endl;
int n = 5;
double z[5] = {2,1,3,5,4};
naivSort2<double, n>(z);    // error!!!
```

Alocare dinamică

- Operatorul **new**:
  - Alocare de obiecte simple:

```
new tip;
new tip(parametri);
```

```
int* pi = new int;
complex* pc = new complex(2., 3.);
string *pstr = new string("Sir alocat dinamic");
```

  - Alocare de tablouri de obiecte:

```
new tip[…];
```

    - Se apelează constructorul implicit

```
double* pt = new double[5];
string *ptstr = new string[30];
int const* q = new int[10];// *q este const
const int* q = new int[10];// la fel
int* const q = new int[10];// q este const
```

Alocare dinamică

- Operatorul **`delete`**:
  - Dealocare obiecte simple:

    `delete pointer;`

    `delete pi;`

  - Dealocare tablouri de obiecte:

    `delete []pointer;`

    `delete []pt;`

    `delete []ptstr;`

  - Pentru tipurile utilizator(clase) la execuția operatorului delete se apelează destructorul clasei

Alocare dinamică: Exemplu

```
struct matrice {
    void init_matrice(int d1,
   int d2);
    void free_mem();
    int  dim1() const {
   return(s1); }
    int  dim2() const {
   return(s2); }
    int**  p;
    int    s1, s2;
```

Alocare dinamică: Exemplu

```
void matrice::init_matrice(int d1, int d2)
{
   s1 = d1;   s2 = d2;
   if(d1 <= 0 || d2 <= 0)
        throw "dimensiune gresita \n";
   p = new int*[s1];
   if(p == 0)
        throw "alocare imposibila \n";
   for (int i = 0; i < s1; ++i){
      p[i] = new int[s2];
      if(p[i] == 0)
              throw "alocare imposibila \n";
}

void matrice::free_mem()
{
   for (int i = 0; i < dim1(); ++i)
      delete p[i];
   delete []p;
}
```

Alocare dinamică: Exemplu

```
istream& operator >> (istream& inp, matrice m)
  {
    for(int i = 0; i<m.dim1(); ++i)
    {
          for(int j = 0; j<m.dim2(); ++j)
                inp >> m.p[i][j];
    }
    return inp;
  }
ostream& operator << (ostream& out, matrice m)
  {
    for(int i = 0; i<m.dim1(); ++i)
    {
          for(int j = 0; j<m.dim2(); ++j)
                out << m.p[i][j] << ' ';
          out << endl;
    }
    return out;
  };
```

POO, o trecere în revistă

- Clase şi obiecte, Mesaje
- Relația de moştenire, Ierarhii de clase
  - Relația de generalizare
  - Relația de specializare
- Relația de asociere
- Relația de compoziție
- Polimorfism

Caracteristicile unui limbaj OOP

- **Încapsulare cu ascunderea datelor**: distincție între starea şi comportarea internă a unui obiect vis-à-vis de starea şi comportarea externă
- **Extensibilitate la nivel de tip**: tipuri definite de utilizator
- **Moştenire**: crearea de noi tipuri prin reutilizarea descrierii tipurilor existente
- **Polimorfism cu legare dinamică**: obiectele sunt responsabile de interpretarea invocarii functiilor

Clase, Obiecte

- **Clasă**: Implementare ADT. Clasa defineste:
  - Atribute – structura de date a ADT
  - Metode – implementarea structurii de date si a operațiilor
  - Instanțele claselor se numesc **obiecte:** clasa defineşte **proprietățile** (atribute) şi **comportarea** (metode) obiectelor
- **Obiect:** instanță a clasei. Este identificat în mod unic prin **nume**
  - defineşte o **stare** care este reprezentată de valorile atributelor sale la un moment dat
  - **comportarea** obiectului este definită de mulțimea metodelor ce pot fi aplicate lui, metode ce-i pot schimba starea
- Un **program:**
  - O colecție de obiecte
  - Interacțiuni între obiecte

Mesaje

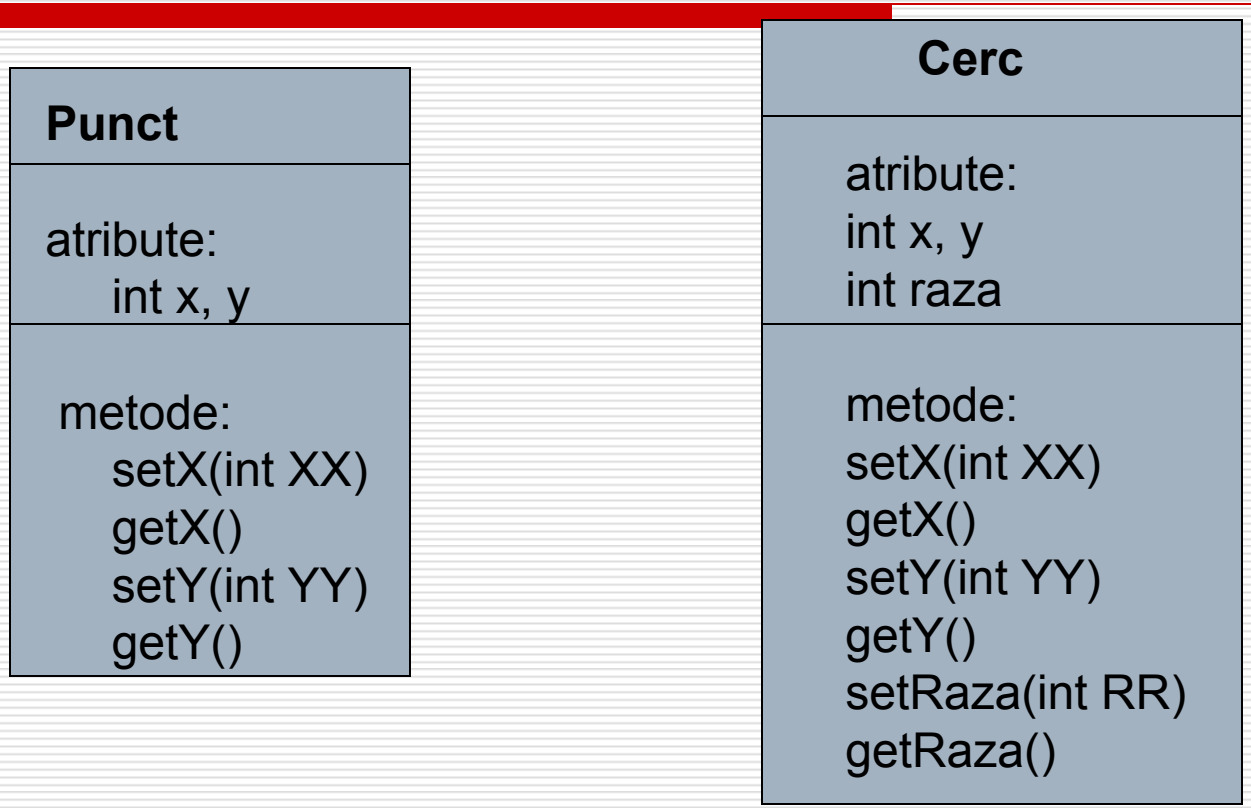
- Într-un program POO sunt create obiecte, sunt distruse obiecte, obiectele interacționează prin transmiterea de mesaje
- Mesaj: i se cere unui obiect să invoce una din metodele clasei din care face parte. Un mesaj conține:
  - Numele metodei
  - Argumentele metodei
- Transmiterea mesajului unui obiect pentru a fi invocată o metodă este similar cu apelul unei funcții; deosebirea este ca un mesaj poate sa se termine cu un eşec dacă metoda invocata nu este cunoscuta de obiect

```
intreg i;      // se creaza obiectul i
i.set_val(5); // mesaj catre obiectul i
```

Mesaje

- Fie definite clasele: **`Facultate, Profesor, Student, Disciplina, Examen.`**
- Exemple de mesaje:
  - facultatea FII comunică profesorului P să examineze studenții înscrişi la examenul E.
  - profesorul P examinează studenții înscrişi la examenul E şi comunică acestuia notele acordate.
  - facultatea FII comunică examenului E să afişeze notele

Relații

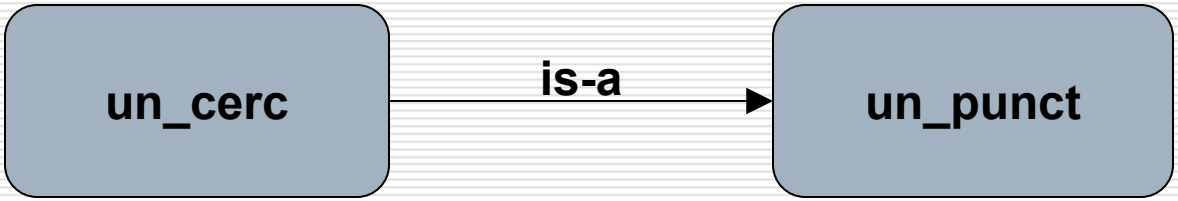


- Comparați definițiile celor două clase

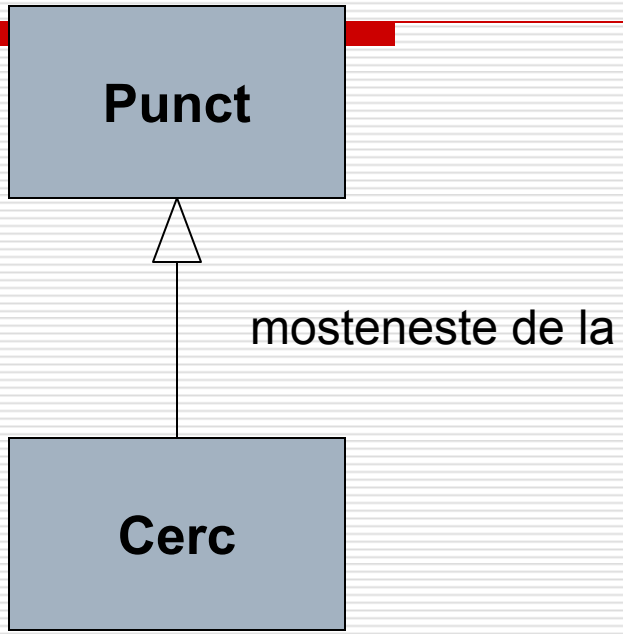
Relații

- Clasa **`Cerc`** este în relația "a-kind-of" cu clasa **`Punct`**

- Un obiect din clasa **`Cerc`** este în relația "is-a" cu un obiect din clasa **`Punct`**

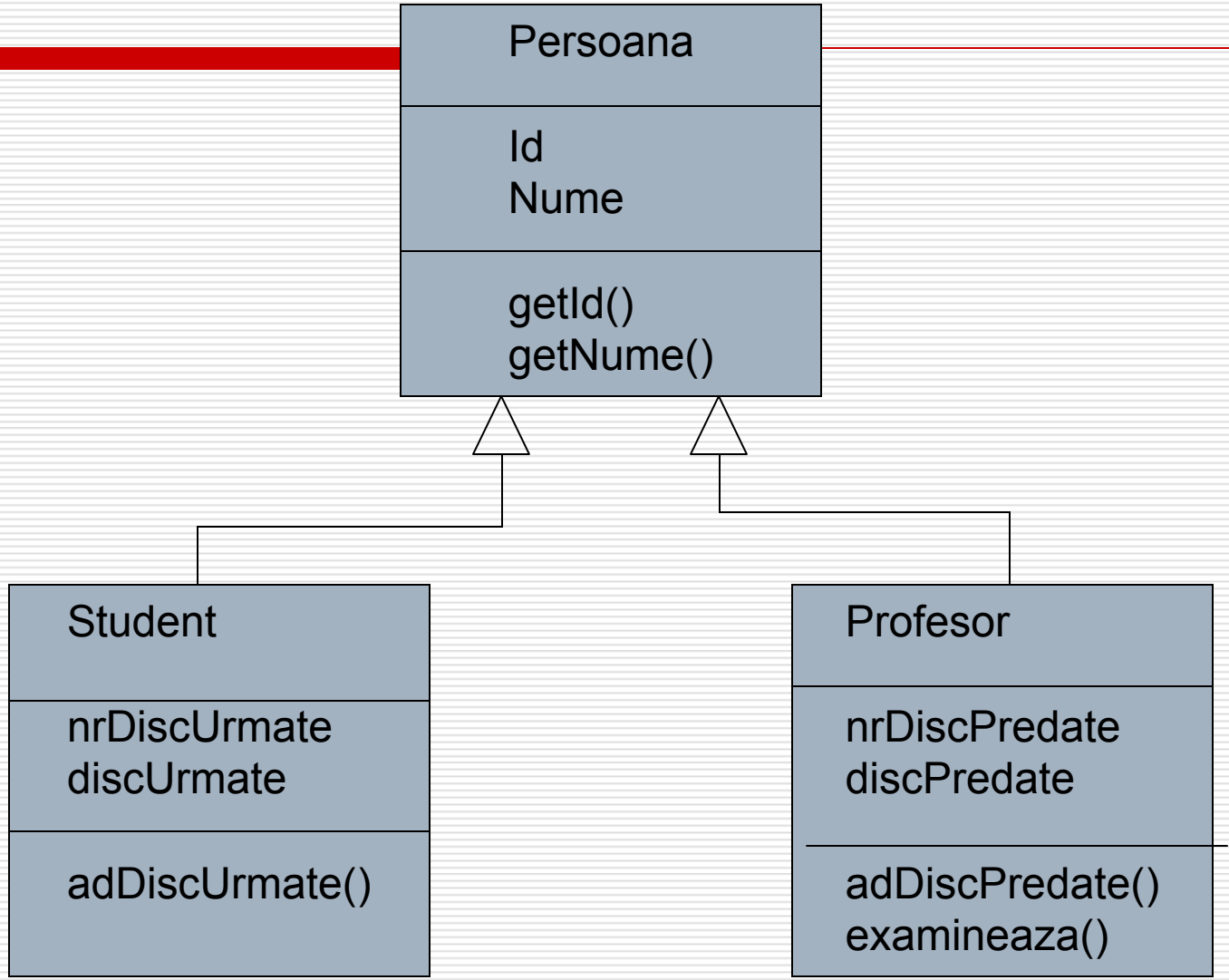


Specializare/Generalizare



- Clasa **`Cerc`** este o **specializare** a clasei **`Punct`**. Ea moşteneşte toate elementele clasei **`Punct`**, date şi metode, fară ca acestea să mai fie definite. **`Cerc`** foloseşte datele si metodele deja definite la **`Punct`**
- Clasa **`Punct`** este o **generalizare** a clasei **`Cerc`**.
- Un cerc este un punct, deci lui i se pot aplica toate metodele din **`Punct`**

Generalizare / Specializare



Moştenire - definiție

- ***Moştenirea*** *este mecanismul care permite unei clase A să reutilizeze datele şi metodele unei clase B. Spunem ca A moşteneşte de la B. Obiectele clasei A au acces la atributele şi metodele clasei B fără a le redefini*

- *Dacă clasa A moşteneşte de la clasa B atunci B se numeşte* ***superclasă (clasă părinte)*** *lui A iar A* ***subclasă (clasă copil, clasă derivată)*** *a lui B.* Obiectele unei subclase pot fi utilizate oriunde sunt îndreptățite să apară obiectele superclasei sale: obiectele subclasei partajează aceleaşi atribute şi aceeaşi comportare *ca* şi obiectele superclasei