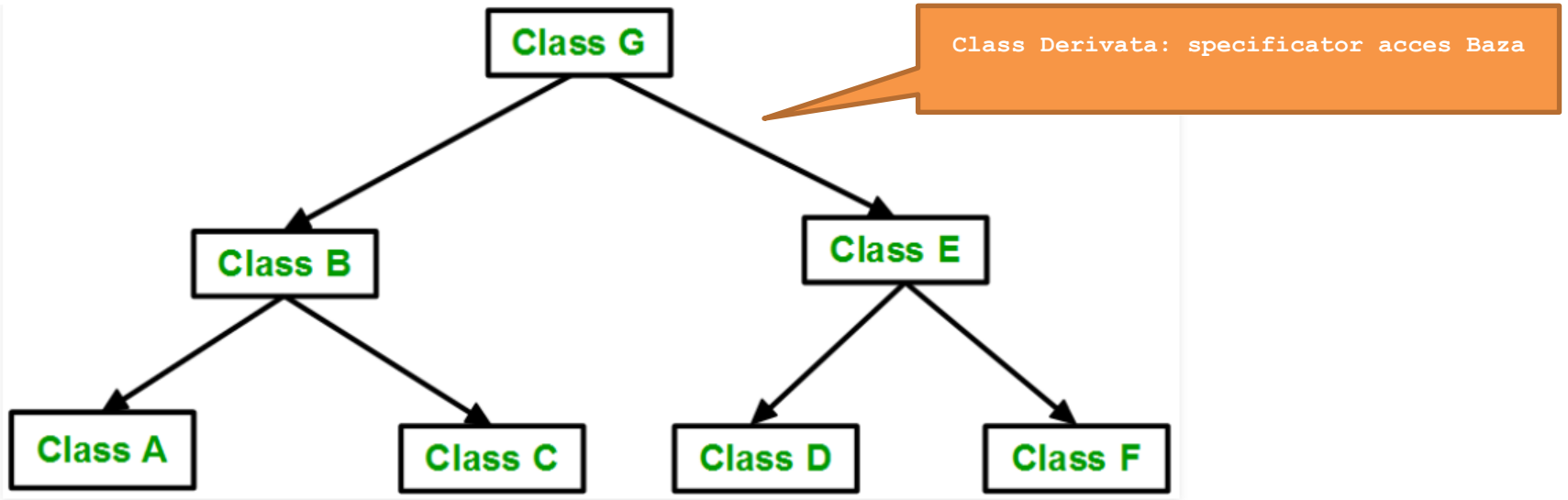
PROGRAMAREA ORIENTATĂ OBIECT C++

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

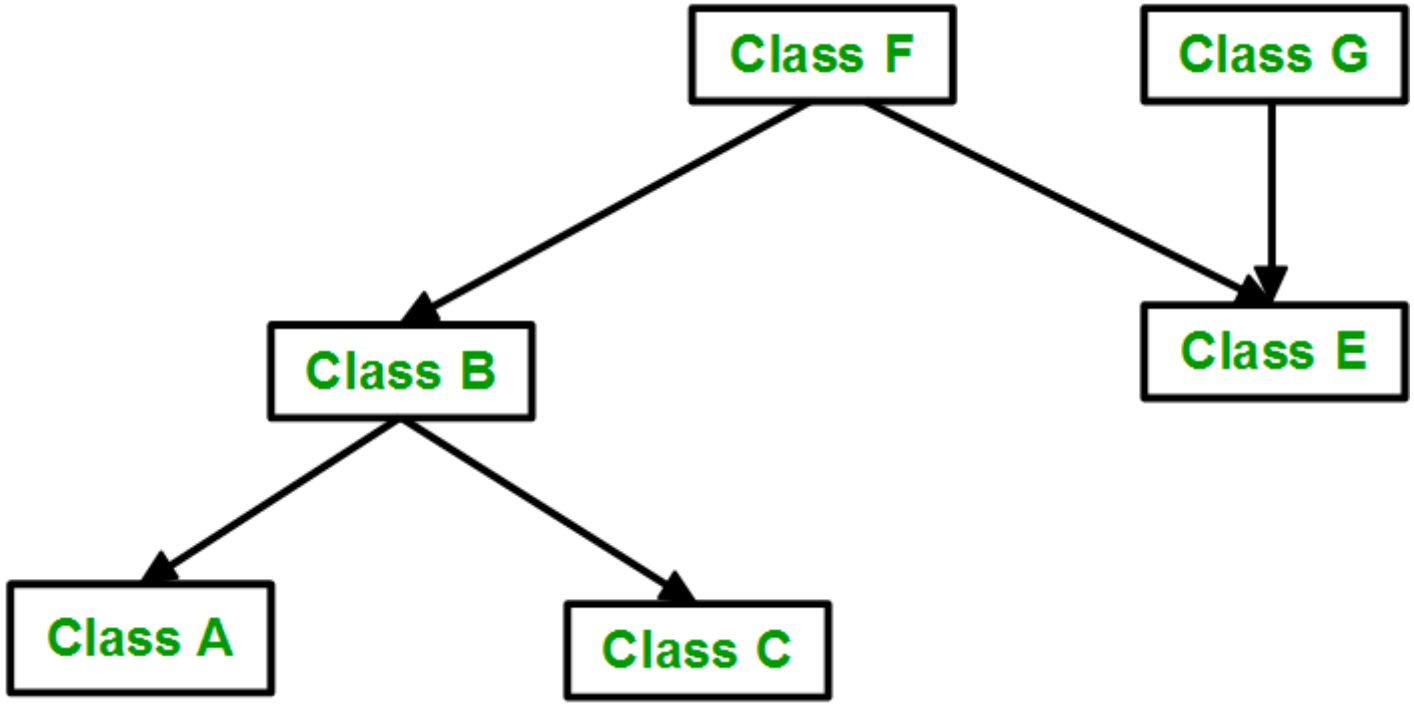
Universitatea Titu Maiorescu

➢În limbajul C++ există două tipuri de moșteniri:

- **Moștenirea singulară**:
  - orice clasă din ierarhia de clase are o singură clasă de bază directă
  - ierarhia de clase se poate reprezenta printr-un graf aciclic (arbore)



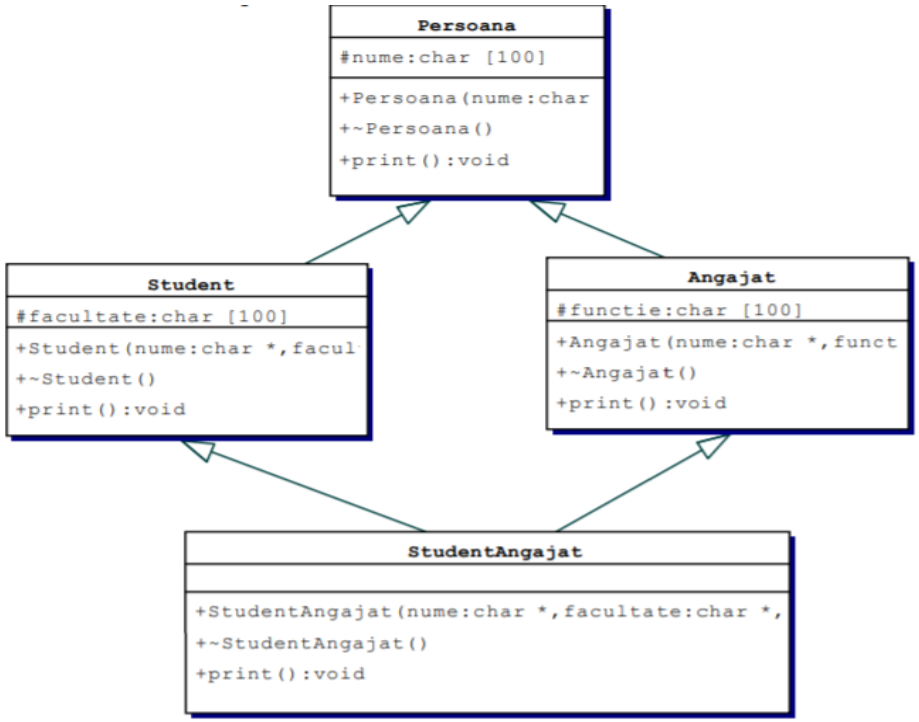
- **Moștenirea multiplă**
  - o clasă din ierarhia de clase se derivează din mai multe clase de bază
  - ierarhia se reprezintă printr-un graf orientat



- Sintaxa moștenire multiplă

```
class Derivata:mod_acces1 Baza1, mod_acces Baza2, ...,
                mod_accesn Bazan
```

- Exemplu



```
class Persoana
{
protected:
    char nume[100];
public:
    Persoana(char *nume)
    {
        strcpy(this->nume, nume);
        cout<<"Apel constructor Persoana\n";
    }
    ~Persoana()
    {
        cout<<"Apel destructor Persoana\n";
    }
    void afisare()
    {
        cout<<"Nume:"<<nume<<endl;
    }
};
```

```
class Student:public Persoana
{
protected:
    char facultate[100];
public:
    Student(char *nume, char *facultate)
                :Persoana(nume)
    {
        strcpy(this->facultate, facultate);
        cout<<"Apel constructor Student\n";
    }
    ~Persoana()
    {
        cout<<"Apel destructor Student\n";
    }
    void afisare()
    {
        Persoana::afisare();
        cout<<"Facultate:"<<facultate<<endl;
    }
};
```

```
class Angajat: public Persoana
{
protected:
    char functie[100];
public:
    Angajat(char *nume, char *functie):
        Persoana(nume)
    {
        strcpy(this->functie, functie);
        cout<<"Apel constructor Angajat\n";
    }
    ~Angajat()
    {
        cout<<"Apel destructor Angajat\n";}
    void afisare()
    {
        Persoana::afisare();
        cout<<"Functie:"<<functie<<endl;
    }}
```

```
class StudentAngajat: public Student,
    public Angajat
{
public:
 StudentAngajat(char *nume,char     *facultate,
char *functie):Student(nume,facultate),
                    Angajat(nume, functie)
    {

cout<<"Apel constructor  StudentAngajat\n";
    }
    ~StudentAngajat()
    {
     cout<<"Apel destructor StudentAngajat\n";
    }
    void afisare();
};
}
```

```
int main()
{
 StudentAngajat sa("Mihai", "Informatica","programator");
  sa.afisare();
}
```

Undefined reference to StudenAngajat::afisare()

➢ Problemă:

- Un obiect din clasa derivată `StudentAngajat` va conţine membrii clasei `Persoana` de două ori, o dată prin clasa `Student` şi o dată prin clasa `Angajat`.
- Se pot reprezenta părţile distincte ale unui astfel de obiect derivat (`StudentAngajat`):

Persoana din Student ..... Student
Persoana din Angajat ..... Angajat
StudentAngajat

➢ Problema diamantului

- O clasă derivată din mai multe clase de bază, poate să preia de două sau mai multe ori date și/sau comportament dintr-o clasa care a fost extinsă în clasele sale de bază!!!!

➢ Soluție

- Se pot elimina ambiguităţile, respectiv erorile de compilare prin calificarea variabilei cu domeniul clasei căreia îi aparţine:

```
StudentAngajat sa("Mihai", "Informatica","programator");
sa.Student::afisare();
sa.Angajat::afisare();
```

- Totuși, se vor apelea de mai multe ori constructorii claselor de bază, respectiv destructorii!!!

➢Alternativă: Clase virtuale

- O altă soluţie pentru eliminarea ambiguităţilor în moştenirile multiple presupunea impunerea unei singure copii a clasei de bază în clasa derivată.

- Astfel, clasă care produce copii multiple prin moştenire indirectă (clasa `Persoana`, în exemplul de mai sus) să fie declarată clasă de bază de tip virtual în clasele care o introduc în clasa cu moştenire multiplă.

```
class L { public: int x; };
class A : virtual public L { /* */ };
class B : virtual public L { /* */ };
class D : public A, public B { /* */ };
```

- Rezultat: data membru `x` se include o singură dată la nivelul clasei derivate D.

➢ Observații

- O clasă poate avea atât clase de bază vrituale, cât și clase de bază non-virtuale

- Constructorii claselor de bază virtuale sunt apelați înaintea constructorilor claselor de bază non virtuale

- Dacă o clasă are mai multe clase de bază virtuale, atunci constructorii claselor virtuale sunt apelați în ordinea declarării lor!!!

- **Polimorfismul** este capacitatea unor entități de a lua forme diferite.

- Etimologia *polimorm Poly = multe + Morfm =forma*

- Este unul din conceptele esențiale din POO

- Există două tipuri de polimorfism
  - Polimorfismul parametric (supraîncărcarea) – mecanismul prin care putem defini o metodă cu același nume în aceași clasă, funcții care trebuie să difere prin numărul și/sau tipul parametrilor. Selecția funcției se realizează la compilate (legarea timpurie (early binging)).

  - Polimorfismul de moștenire (redefinire și funcții virtuale) – mecanismul prin care o metodă din clasa de bază este redefinită cu aceiași parametri în clasele derivate. Selecția funcției se va realiza la rulare (legarea întârziată (late binding,dynamic binding, runtime binding)).

➢Conversia pointerilor între clasa de bază și cea virtuală

- **Upcating**: conversia unui pointer la o clasă derivată în pointer la o clasă de bază a acesteia este implicită, dacă derivarea este de tip public.
- **Downcasting**: Conversia inversă, a unui pointer la o clasă de bază în pointer la derivată nu este admisă implicit.

- O astfel de conversie se poate forţa explicit prin operatorul de conversie cast.
- Rezultatul unei astfel de conversii este însă nedeterminat, de cele mai multe ori provocând erori de execuţie!!!!

- Considerăm următoarea ierahie de clase:

```
class B;
class D: public B{    };

D d;
B* pb = &d;

B ob;
D* pd = &ob;

D* pd =(D*)&ob;
```

Upcating -> este o conversie corectă

Downcasting -> eroare la compilare

Upcating -> este o conversie corectă

➢ Funcții virtuale

- Considerăm următoarea ierarhie de clase

```
class B
{
  public:
void f()
{
cout<<"f din   baza"<<endl;}
        .....
}
```

```
class B:public D
{
  public:
  void f()
{
cout<<"f din   derivata"<<endl;}
        .....
}
int main()
{
    B *ob = new D();
    ob->f();
}
```

Se aplelază metoda după tipul pointerului

f din baza

- Obiectul ob are două tipuri de dată:
  - un tip declarat: B
  - un tip real: D

- Dacă un obiect derivat este definit printr-un pointer de tipul clasei de bază, atunci metodă refedinită în clasa derivată, se apelează în raport cu tipul pointerului, respectiv, din clasa de bază.

- Dacă o metodă este definită ca o metodă virtuală în clasa de bază şi redefinită în clasele derivate, la apelul acesteia ca funcţie membră a unui obiect pentru care se cunoaşte un pointer, se selectează funcţia după tipul obiectului, nu al pointerului!!!!

- Sunt posibile mai multe situaţii:

  - Dacă obiectul este de tip clasă de bază nu se poate folosi un pointer la o clasă derivată

  - Dacă obiectul este de tip clasă derivată şi pointerul este pointer la clasă derivată, se selectează funcţia redefinită în clasa derivată respectivă.

  - Dacă obiectul este de tip clasă derivată, iar pointerul folosit este un pointer la o clasă de bază a acesteia, se selectează funcţia redefinită în clasa derivată corespunzătoare tipului obiectului.

- Exemplu

```
class B
{
  public:
void f()
    {cout<<”f din  B\n”;}
virtual void g()
   {cout << "g() din B"; } }
```

```
class B:public D
{
  public:
  void f()
     {cout<<”f din  D\n”;}
 void g()
   {cout << "g() din D"; }
     }
```

```
B* pb = new B;
D1* pd1 = new D1;

pb->f();
pb->g();

pd1->f();
pd1->g();
```

```
f() din B
g() din B
f() din D
g() din D
```

```
B* pb1 = pd1;

pb1->f();
pb1->g();
```

```
f() din B
g() din D
```

- Avantaje

  - Se pot defini colecții de obiecte ale aceleiași ierarhi într-o manieră unitară deoarece funcţia apelată virtual este identificată la rulare cu funcţia membră din clasa căreia îi aparţine obiectul

  - Se poate asigura independența implementărilor

- Destructori virtuali
- Considerăm următorul exemplu

```
class B{
public:
  B(){ cout << "Constructor B\n";}
  ~B(){cout << "Destructor B\n";}
};
class D:public B {
public:
  D(){ cout << "Constructor D\n";}
  ~D(){cout << "Destructor D\n";}
};
void main(){
  B* p = new D(); // creaza un obiect D
  delete p; // sterge un obiect B
}
```

```
Constructor B
Constructor D
Destructor B
```

➢ Problemă

- Rămâne ocupată în mod inutil o zonă de memorie heap, respective partea de date a clasei D

➢ Soluție

- Declararea destructorului din clasa de bază de tip virtual

```
class B{
public:
B(){ cout << "Constructor B\n";}
virtual ~B(){cout << "Destructor B virtual\n";}
};
```

```
Constructor B
Constructor D
Destructor D
Desctructor virtual B
```