

Programare II

Limbajul C/C++

CURS 10

Curs anterior

- ❑ STL
 - ❑ Containere
 - ❑ Iteratori
 - ❑ Algoritmi

Cuprins

- ❑ Moștenire simplă
- ❑ Funcții virtuale
- ❑ Polimorfism

Relații între clase

- ❑ Conceptele nu există izolate. Ele **coexistă și interacționează**.
- ❑ La elaborarea modelului obiectual al unei aplicații se disting următoarele etape:
 - ❑ **Identificarea claselor** → corespund conceptelor aplicației - substantive –
 - ❑ **Stabilirea relațiilor dintre clase** → corespunde specificațiilor aplicației – verbe –
- ❑ Tipuri de relații
 - ❑ Asociere o relație în care obiectele unei clase știu de obiectele altei clase (has-a)
 - ❑ **Dependență** o relație în care obiectele unei clase știu de obiectele altei clase (use-a)
 - ❑ **Agregare** o relație parte întreg (is-part-of)
 - ❑ **Compoziție** este similară cu asocierea dar mai strictă (contains)
 - ❑ **Mostenire** (specializare) este o relație de generalizare - specializare (is-a, kind-of)

Moștenire (Inheritance)

❑ Definiție

- ❑ Moștenirea este un mecanism care permite unei clase A să **moștenească atribute și metode** ale unei clase B. În acest caz obiectele clasei A au acces la membrii clasei B fără a fi nevoie să le redefinim

❑ Terminologie

❑ Clasă de bază

- ❑ Clasa care este moștenită

❑ Clasă derivată

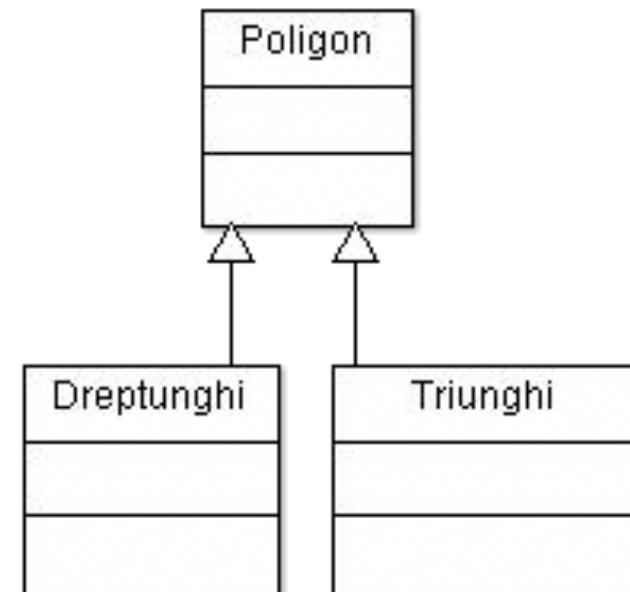
- ❑ O clasă specializată a clasei de bază

❑ Relația „kind-of” nivel de clasă

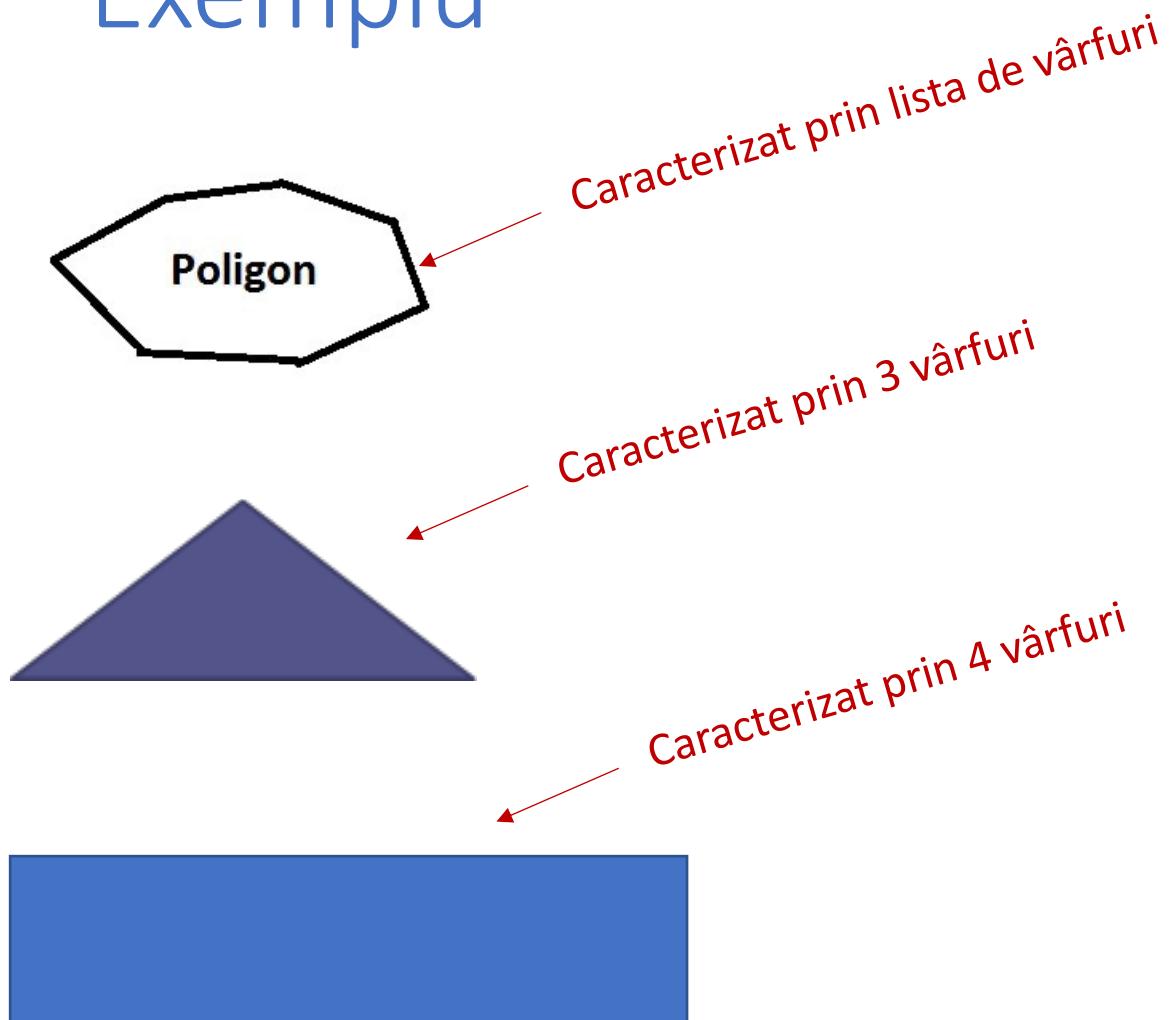
- ❑ Triunghiul este un tip (kind-of) Poligon

❑ Relația „is-a” nivel de obiect

- ❑ Obiectul triunghiRosu este un (is-a) Poligon



Exemplu

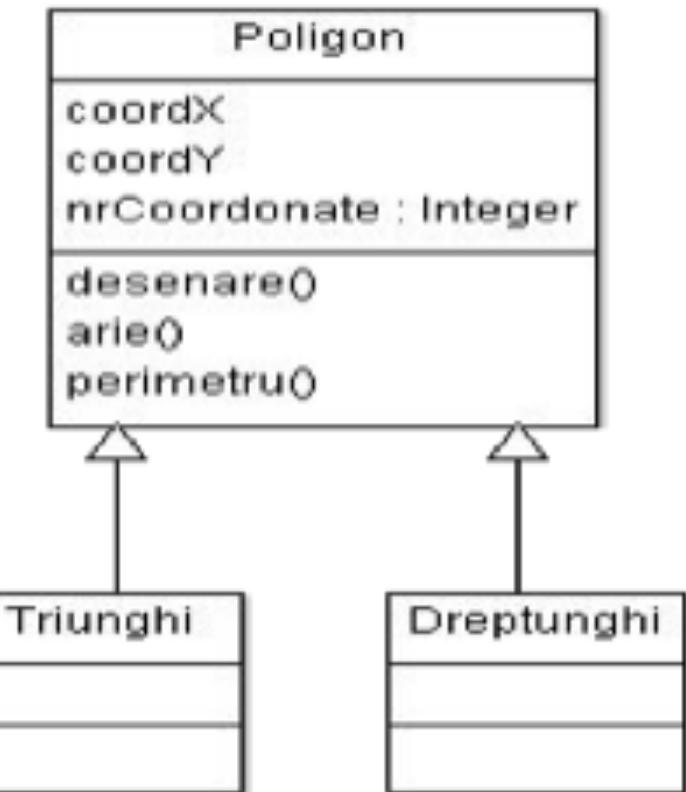
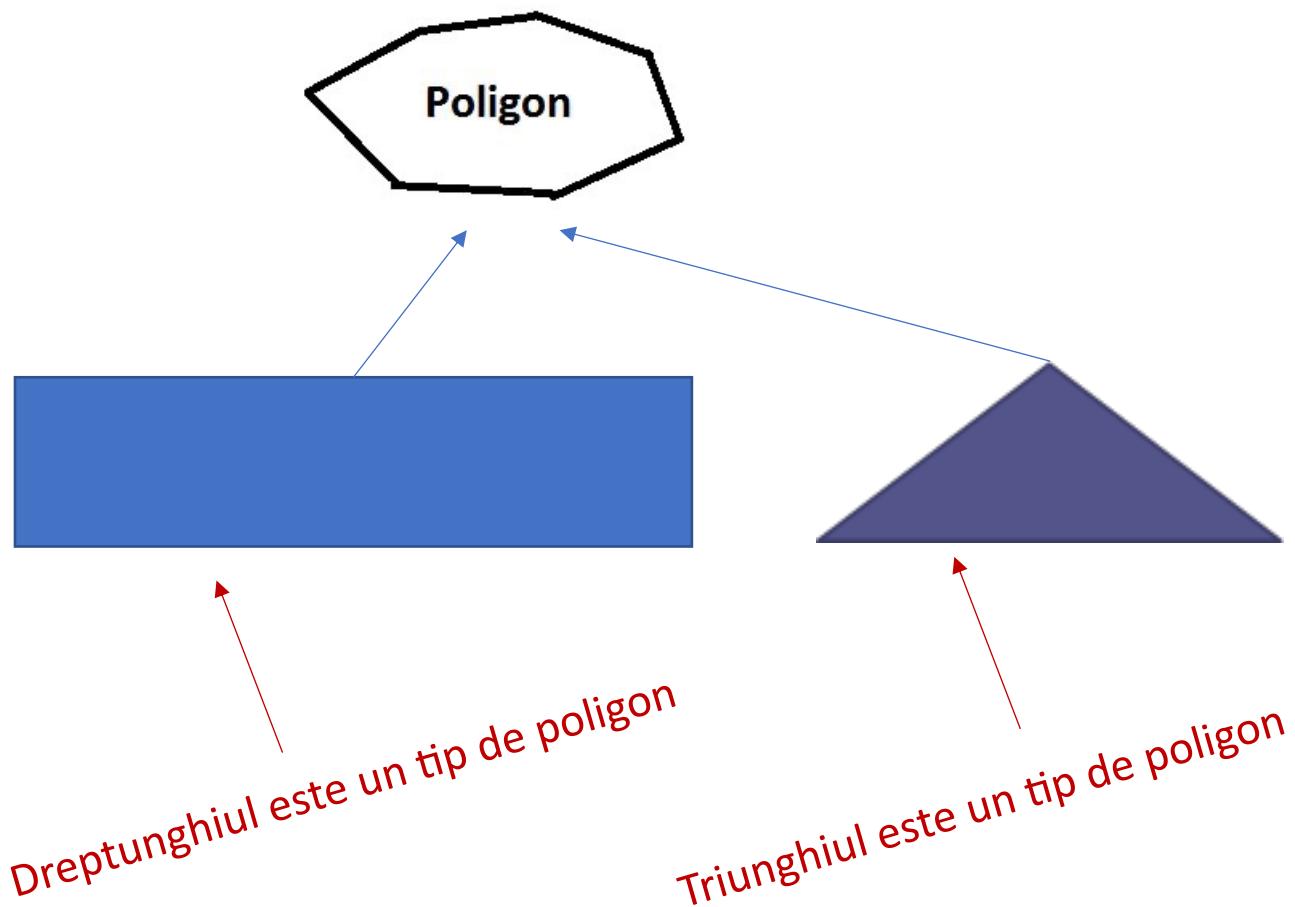


Poligon
coordX
coordY
nrCoordonate : Integer
desenare()

Triunghi
coordX
coordY
desenare()
arie()
perimetru()

Dreptunghi
coordX
coordY
desenare()
arie()
perimetru()

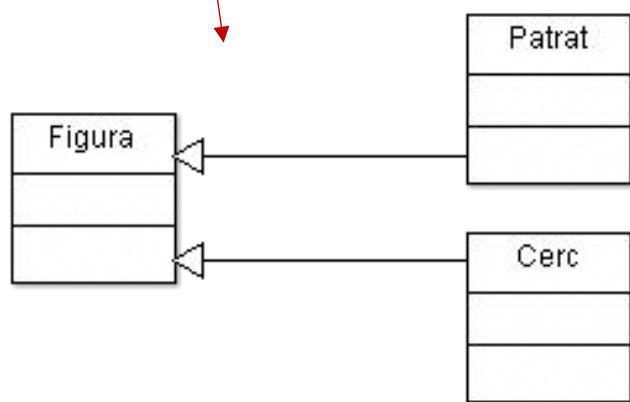
Exemplu



Tipuri de moștenire în C++

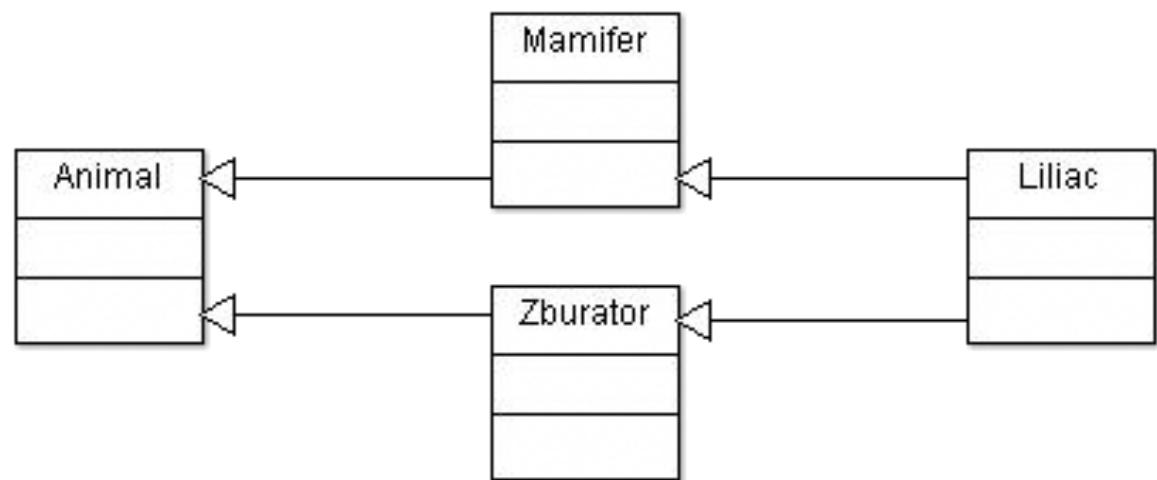
Moștenire Simplă

O singură clasă de bază directă



Moștenire Multiplă

Mai multe clase de bază directe



Definirea unei ierarhii de clase

❑ Sintaxă

```
class NumeleClaseiDerivate : modificatorDeAccess  
NumeleClaseiDeBază
```

❑ unde

- ❑ modificatorDeAcces specifică tipul derivării
 - ❑ private (valoare implicită)
 - ❑ protected
 - ❑ public

❑ Obs

- ❑ Orice clasă poate fi clasă de bază

Modificatori de acces

- ❑ Funcțiile membre ale clasei derivate au acces la membrii publici și protected ai clasei de bază
- ❑ Pentru a controla accesul la membrii clasei de bază sunt folosiți specificatorii de acces

Clasa de bază	Modifierul de acces	Ce se poate accesa în clasa derivată	Ce se poate accesa în exterior
private protected public	private private private	nu este accesibil private private	nu este accesibil nu este accesibil nu este accesibil
private protected public	protected protected protected	nu este accesibil protected protected	nu este accesibil nu este accesibil nu este accesibil
private protected public	public public public	nu este accesibil protected public	nu este accesibil nu este accesibil public

Modificatori de acces

```
class B {  
public:      int a;  
protected:   int b;  
private:     int c;  
};  
  
class D1: B { /*derivare private*/  
    void foo() {  
        int a1=a; //Ok  
        int b1=b; //Ok  
        int c1=c; //eroare: B::c este privat  
    } };  
  
class D2: protected B {  
    void foo() {  
        int a2=a; //Ok  
        int b2=b; //Ok  
        int c2=c; //eroare: B::c este privat  
    } };  
  
class D3: public B {  
    void foo() {  
        int a3=a; //??  
        int b3=b; //??  
        int c3=c; //??  
    } };
```

```
class DD : D1 {  
    void foo();  
};  
  
void DD::foo() {  
    a = 10; // ??  
    b = 20; // ??  
    c = 30; // ??  
}  
int main(int argc, char** argv) {  
    D1 d1; D2 d2; D3 d3;  
  
    int v1=d1.a; //eroare: B::a nu este accesibil  
    int v2=d1.b; //eroare: B::b este protected  
    int v3=d1.c; //eroare: B::c nu este accesibil  
  
    int v4=d2.a; //eroare: B::a nu este accesibil  
    int v5=d2.b; //eroare: B::b este protected  
    int v6=d2.c; //eroare: B::c este privat  
  
    int v7=d3.a; //OK  
    int v8=d3.b; //eroare: B::b este protected  
    int v9=d3.c; //eroare: B::c este privat  
    return 0;  
}
```

Ce se moștenește?

- ❑ În principiu, **fiecare membru** al clasei de bază este moștenit de clasa derivată
 - ❑ Doar cu **diferite permisiuni** de acces
- ❑ Dar, există câteva **excepții**
 - ❑ Constructorii
 - ❑ Destructorii
 - ❑ Operatorul =
 - ❑ Funcțiile friend

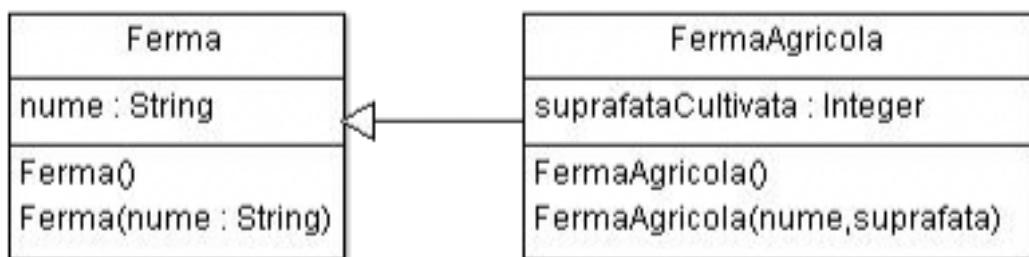
Constructorii și destructorii

❑ Constructori

- ❑ Prima dată se apelează constructorul clasei de bază și apoi constructorul clasei derivate

❑ Destructori

- ❑ Sunt apelați în **ordine inversă** față de constructori, mai întâi se apelează destructorul clasei derivate apoi cel al clasei de bază



Constructorii și deconstructorii

Ferma

```
class Ferma {  
protected:  
    char* nume;  
public:  
    Ferma(){  
        cout << "Ferma: Constructor implicit " << this << endl;  
    }  
    Ferma(char* nume){  
        cout<<"Ferma: Constructor cu parametrii "<<this <<endl;  
        ...  
    }  
    ~Ferma(){  
        cout << "Ferma: Destructor " << this << endl;  
        ...  
    }  
};  
Ce funcții se apelează la execuția  
următorului program?  
int main() { FermaAgricola fa; }
```



Ferma Agricolă

```
class FermaAgricola : public Ferma{  
    int suprafataCultivata;  
public:  
    FermaAgricola(){  
        cout << "FermaAgricola: Constructor implicit " << this << endl;  
    }  
    FermaAgricola(char* nume, int suprafata){  
        cout << "FermaAgricola : Constructor cu parametrii " << this << endl;  
        ...  
    }  
    ~FermaAgricola(){  
        cout << "FermaAgricola: Destructor " << this << endl;  
    }  
};  
Ce funcții se apelează la execuția următorului program?  
int main() { FermaAgricola fa("FermaA", 30); }
```



int main()	Rezultat
FermaAgricola fa;	Ferma: Constructor implicit 0x28ff18 FermaAgricola: Constructor implicit 0x28ff18 FermaAgricola: Destructor 0x28ff18 Ferma: Destructor 0x28ff18

int main()	Rezultat
	FermaAgricola fa("FermaA", 30); Ferma: Constructor implicit 0x28ff18 FermaAgricola: Constructor cu parametrii 0x28ff18 FermaAgricola: Destructor 0x28ff18 Ferma: Destructor 0x28ff18

Constructorii și deconstructorii

Ferma

```
class Ferma {  
protected:  
    char* nume;  
public:  
    Ferma(){  
        cout << "Ferma: Constructor implicit " << this << endl;  
    }  
    Ferma(char* nume){  
        cout << "Ferma: Constructor cu parametrii " << this << endl;  
        ...  
    }  
    ~Ferma(){  
        cout << "Ferma: Destructor " << this << endl;  
        ...  
    }  
};  
Ce funcții se apelează la execuția  
următorului program?  
int main() { FermaAgricola fa; }
```



Ferma Agricolă

```
class FermaAgricola : public Ferma{  
    int suprafataCultivata;  
public:  
    FermaAgricola(){  
        cout << "FermaAgricola: Constructor implicit " << this << endl;  
    }  
    FermaAgricola(char* nume, int suprafata):Ferma(nume){  
        cout << "FermaAgricola : Constructor cu parametri " << this << endl;  
        ...  
    }  
    ~FermaAgricola(){  
        cout << "FermaAgricola: Destructor " << this << endl;  
    }  
};  
Ce funcții se apelează la execuția următorului program?  
int main() { FermaAgricola fa("FermaA", 30); }
```



int main()	Rezultat
FermaAgricola fa;	Ferma: Constructor implicit 0x28ff18 FermaAgricola: Constructor implicit 0x28ff18 FermaAgricola: Destructor 0x28ff18 Ferma: Destructor 0x28ff18

int main()	Rezultat
	FermaAgricola fa("FermaA", 30); Ferma: Constructor cu parametri 0x28ff18 FermaAgricola: Constructor cu parametri 0x28ff18 FermaAgricola: Destructor 0x28ff18 Ferma: Destructor 0x28ff18

Apel explicit al constructorului
cu parametrii ai clasei Ferma

Constructorul de copiere

- Clasa derivată nu are definit un constructor de copiere => se apelează constructorul de copiere creat de compilator
- Clasa derivată are definit un constructor de copiere => se apelează constructorul de copiere definit
 - ex. FermaAnimale a ("Ferma 1", 20), copie(a);

Cum se poate modifica codul astfel încât să se realizeze și o copie a atributelor clasei de bază?



```
FermaAgricola(const FermaAgricola& f) {  
    cout << "FermaAgricola: Constructor"  
        << "de copiere" << this << endl;  
}
```

```
Ferma: Constructor implicit 0x28ff10  
FermaAgricola: Constructor de copiere 0x28ff10  
FermaAgricola: Destructo 0x28ff10  
Ferma: Destructo 0x28ff10
```

Constructorul de copiere

Apelarea constructorului supraclassei

```
FermaAgricola(const  
FermaAgricola& f) :Ferma(f.nume) {  
    cout << "FermaAgricola: " <<  
        "Constructor de copiere" <<  
        this << endl;  
}
```

Ferma: **Constructor cu parametri 0x28ff10**

FermaAgricola: **Constructor de copiere 0x28ff10**

FermaAgricola: Destructor 0x28ff10

Ferma: Destructor 0x28ff10

Apelarea constructorului de copiere al supraclassei

```
FermaAgricola(const  
FermaAgricola& f) :Ferma(f) {  
    cout << "FermaAgricola: " <<  
        "Constructor de copiere" <<  
        this << endl;  
}
```

Ferma: **Constructor de copiere 0x28ff10**

FermaAgricola: **Constructor de copiere 0x28ff10**

FermaAgricola: Destructor 0x28ff10

Ferma: Destructor 0x28ff10

Supraîncărcarea operatorilor

- ❑ Funcțiile operator membre ale clasei de bază sunt moștenite de clasele derivate și pot fi redefinite în clasele derivate
- ❑ EXCEPTIE
 - ❑ operatorul =()
 - ❑ Dacă este definit în clasa derivată, el este apelat, dar nu se mai apelează cel din clasa de bază
 - ❑ Dacă nu este definit în clasa derivată, se apelează implicit copierea pentru atributele clasei derivate și operatorul =() definit în clasa de bază

Conversii de tip

Permise

- D->B ($f = fa$)
- *D->*B ($pf = pfa$)
- *B -> (B*)*D ($pf = (\text{Ferma}^*)pfa$)

```
FermaAgricola fa("Ferma Herneacova", 30), *pfa;  
Ferma f("ferma2"), *pf;  
FermaAgricola fa2(fa); fa2.setSuprafata(100);
```

```
cout << "Conversie implicita FermaAgricola -> Ferma\n";  
f = fa;  
cout << f << endl;
```

```
cout << "Conversie implicita FermaAgricola -> *Ferma\n";  
pf = &fa2;  
cout << *pf << endl;
```

```
cout << "Conversie explicita *Ferma -> *FermaAgricola\n";  
pfa = (FermaAgricola*)pf;  
cout << *pfa << endl;
```

Nepermise

- B->D ($fa = f$)
- *B->*D ($pfa = pf$)

Clasa de bază nu initializează
atributele specifice clasei derivate
(conversie explicită la Ferma a unui
obiect de tip FermaAnimale)

	Output
FermaAgricola fa("Ferma Herneacova", 30), *pfa; Ferma f("ferma2"), *pf; FermaAgricola fa2(fa); fa2.setSuprafata(100);	Ferma Herneacova
cout << "Conversie implicita FermaAgricola -> Ferma\n"; f = fa; cout << f << endl;	Ferma Herneacova
cout << "Conversie implicita FermaAgricola -> *Ferma\n"; pf = &fa2; cout << *pf << endl;	Ferma Herneacova
cout << "Conversie explicita *Ferma -> *FermaAgricola\n"; pfa = (FermaAgricola*)pf; cout << *pfa << endl;	Ferma Herneacova 770887

Redefinirea funcțiilor membre

- ❑ Clasele derivate pot redefini funcțiile definite în clasele de bază
 - ❑ Restricții
 - ❑ Trebuie să aibă același specificator de vizibilitate
 - ❑ Trebuie să aibă aceeași listă de parametrii

Clasa Ferma

```
void Ferma::afisare() {  
    cout << nume;  
}
```

Clasa FermaAgricola

```
void FermaAgricola::afisare() {  
    cout << nume << " cu suprafata "  
        << suprafata;  
}
```

Redefinirea funcțiilor membre

- ❑ Funcțiile redefinite în clasele derivate ascund funcțiile din clasele de bază
 - ❑ Funcțiile din clasa de bază pot fi accesate folosind operatorul de rezoluție

❑ Exemplu

```
void FermaAgricola::modificareNume(char * nume) {  
    cout << "Ferma agricola: '";  
    afisare();  
    if (nume!=NULL){  
        this->nume = new char[strlen(nume)+1];  
        strcpy(this->nume, nume);  
        cout << "' isi schimba numele in '";  
        Ferma::afisare();  
        cout << "'\n"  
    }  
}
```

Apel funcție afișare() din clasa
FermaAgricola

Apel funcție afișare() din clasa Ferma

❑ Output

- ❑ FermaAgricola fa("Ferma Herneacova", 30); fa.modificareNume ("Ferma Izvin")
- ❑ Ferma agricola: 'ferma Herneacova cu suprafata 30' isi schimba numele in 'Ferma Izvin'

Functii virtuale

Clasa Ferma

```
void Ferma::afisare() {  
    cout << nume;  
}
```



Care este rezultatul executiei urmatorului cod?

```
int main() {  
    Ferma *pf;  
    FermaAgricola fa("Ferma Buzias", 300); pf = &fa;  
    pf->afisare();  
    return 0;  
}
```

Clasa FermaAgricola

```
void FermaAgricola::afisare() {  
    cout << nume << " cu suprafata "  
        << suprafata;  
}
```

Răspunsul este a) deoarece funcția afisare() este apelată din clasa Ferma, chiar dacă în variabila ff referă la un obiect de tip FermaAgricola

- a) Ferma: Ferma Buzias
- b) Ferma Agricola: Ferma Buzias cu suprafata 300

Functii virtuale

❑ Solutii

- ❑ Adăugarea unui câmp care specifică tipul obiectului
- ❑ Funcții virtuale

❑ Tipuri de legături

- ❑ **Statică** (early binding)
 - ❑ Versiunea funcției apelate se stabilește în momentul compilării
- ❑ **Dinamică** (late binding)
 - ❑ Versiunea funcției apelate se stabilește în timpul execuției programului

Functii virtuale

□ Sintaxă

virtual prototipFunctie;

□ Caracteristici

- Atributul virtual este **moștenit**
- Este un tip special de funcție care **determină tipul derivat corespunzător** pentru o funcție cu același prototip doar în **cazul variabilelor de tip pointer**
- Specificarea cuvântului „**virtual**” în fața funcției
- Sunt funcții **membre nestatice**
- Redefinirea și redeclararea funcțiilor virtuale în clasele derivate nu este obligatorie
- **Constructori nu** pot fi funcții virtuale
- Redefinirea sau schimbarea prototipului este acceptabilă doar dacă se modifică valoarea de return

Functii virtuale

Clasa Ferma

```
class Ferma {  
    ...  
    virtual void afisare();  
    ...  
};  
void Ferma::afisare() {  
    cout << nume;  
}
```



Care este rezultatul executiei urmatorului cod?

```
int main() {  
    Ferma *pf;  
    FermaAgricola fa("Ferma Buzias", 300); pf = &fa;  
    pf->afisare();  
    return 0;  
}
```

Clasa FermaAgricola

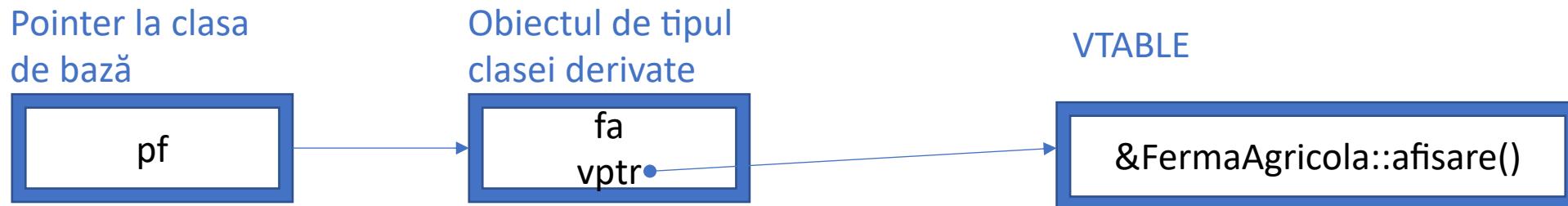
```
void FermaAgricola::afisare() {  
    cout << nume << " cu suprafata "  
        << suprafata;  
}
```

Răspunsul este b)

- a) Ferma: Ferma Buzias
- b) Ferma Agricola: Ferma Buzias cu suprafata 300

Functii virtuale

❑ Ce se întâmplă intern?



❑ Pentru realizarea legarea dinamica (late binding)

- ❑ compilatorul creează un **VTABLE** pentru fiecare **clasă** care conține funcții virtuale
- ❑ Adresa funcției virtuale este adăugată în aceste tabele

❑ Când se creează un **obiect**

- ❑ Compilatorul adaugă un pointer numit vpointer care pontează spre **VTABLE** obiectului
- ❑ Când funcția este apelată compilatorul rezolvă corect legătura prin intermediul **vpointer**

Destructori virtuali

- ❑ Ar trebui ca destructori claselor bază să fie declarați virtuali? De ce da sau de ce nu?
- ❑ **Da!** Trebuie să ștergem întotdeauna și pointerii din subclase (altfel apare riscul de **memory leaks**)
- ❑ Dacă într-un pointer de tip clasa de bază stocăm un obiect de tipul unei clase derivate și ștergem pointerul de tipul de bază destructorul clasei derivate nu va fi apelat dacă destructorul clasei de bază nu este virtual

Destructori virtuali

```
class Basel {  
public:  
    ~Base1() { std::cout << "~Base1()\n"; }  
};  
class Derived1 : public Basel {  
public:  
    ~Derived1() { std::cout << "~Derived1()\n"; } }  
;  
class Base2 {  
public:  
    virtual ~Base2() { std::cout << "~Base2()\n"; }  
};  
class Derived2 : public Base2 {  
public:  
    ~Derived2() { std::cout << "~Derived2()\n"; }  
};
```

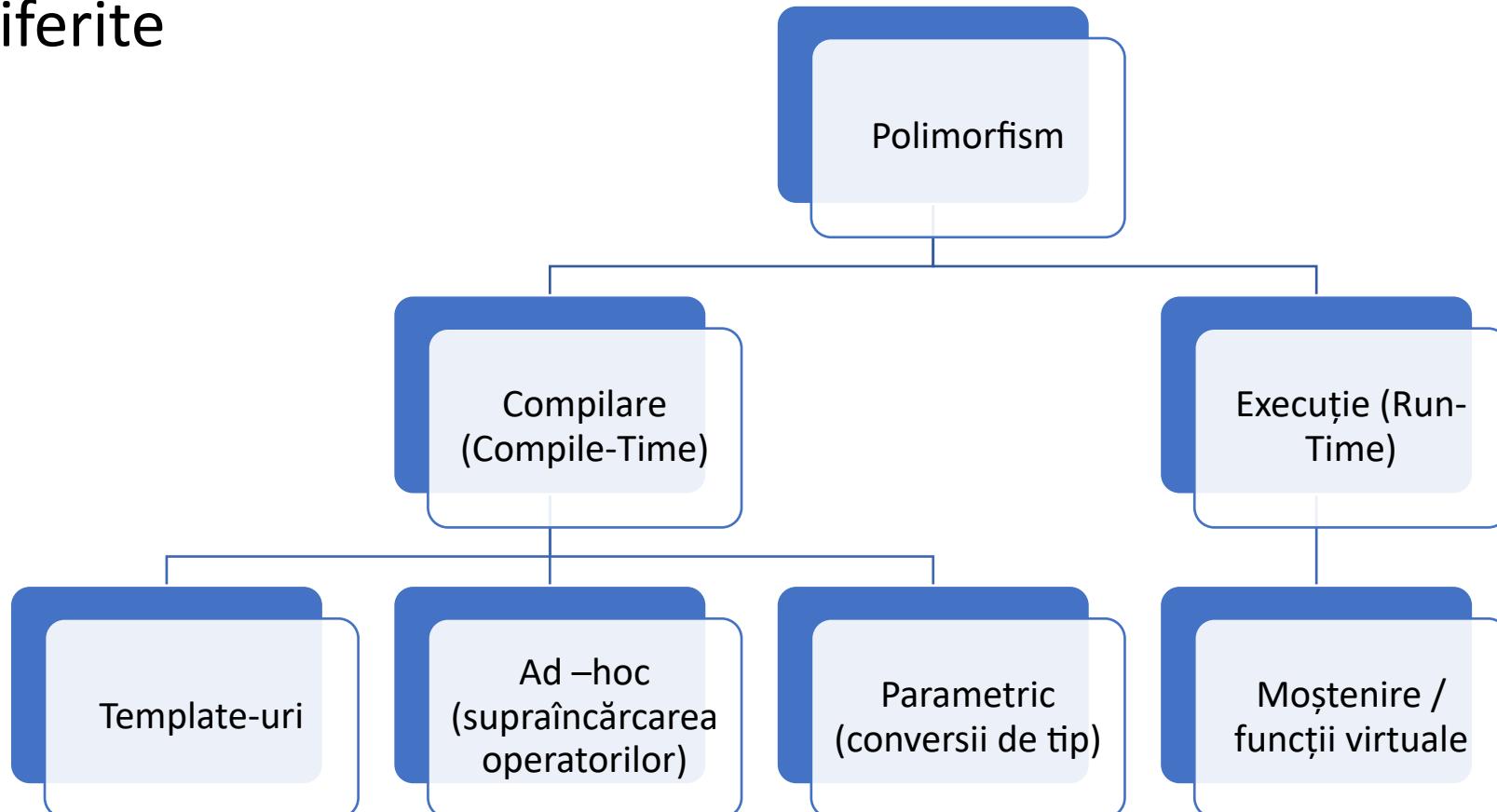
```
int main() {  
    Basel* bp = new Derived1;  
    delete bp;  
  
    Base2* b2p = new Derived2;  
    delete b2p;  
}
```

Rezultatul execuției codului este?
~Base1()

Rezultatul execuției codului este?
~Derived2()
~Base2()

Polimorfism

- Polimorfismul este abilitatea de a folosi o metodă sau un operator în moduri diferite



Sumar

- Moștenire
 - O metodă care permite **refolosirea codului**
 - Folosirea **moștenirii publice** pentru polimorfism
 - Folosirea **moștenirii private** pentru încapsulare
- Polimorfism
 - Folosirea de **pointeri la clasele de bază**
 - Legătura statică/**dinamică**



ÎNTREBĂRI