Programare II
Programare Orientată Obiect

Curs 7

Curs anterior

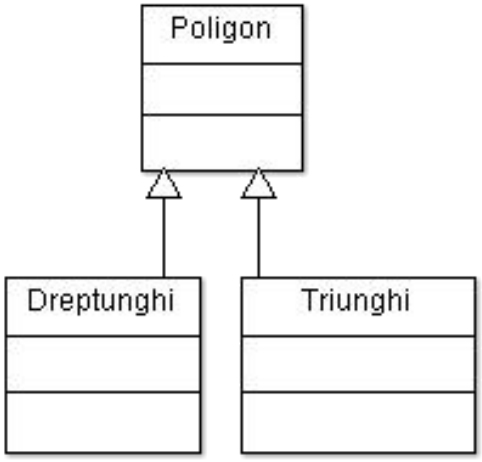
- Supraîncărcarea operatorilor unari
- Supraîncărcarea operatorilor
  - Conversii de tip
- Excepţii
  - Aruncarea excepţiilor
  - Prinderea excepţiilor

Curs curent

- Moştenire
- Clase derivate
- Modificatori de acces
- Constructori şi destructori
- Supraîncărcarea operatoriilor
- Supradefinirea funcţiilor
- Funcţii virtuale
- Destructori virtuali
- Polimorfism

Moştenire (inheritance)

- Definiţie
  - Moştenirea este un mecanism care permite unei clase A să moştenească atribute şi 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 moştenită
  - Clasă derivată
    - O clasă specializată a clasei de bază
  - Relaţia „kind-of” nivel de clasă
    - Triunghiul este un tip (kind-of) Poligon
  - Relaţia „is-a” nivel de obiect
    - Obiectul triungiRosu este un (is-a) Poligon



Exemplu

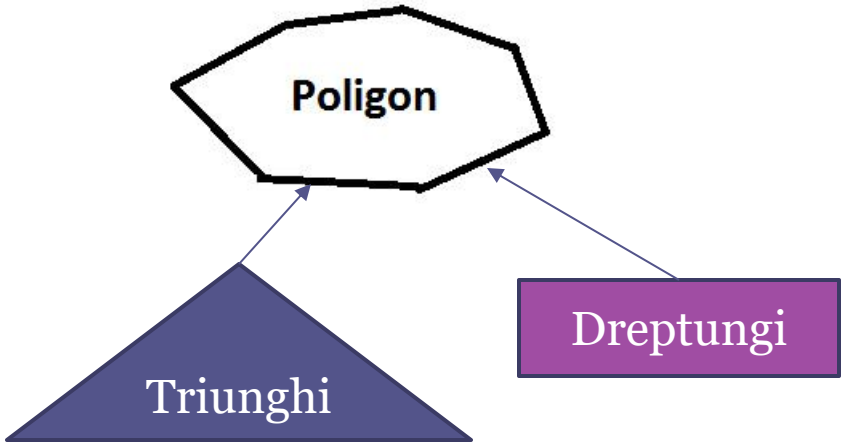
Dreptungi

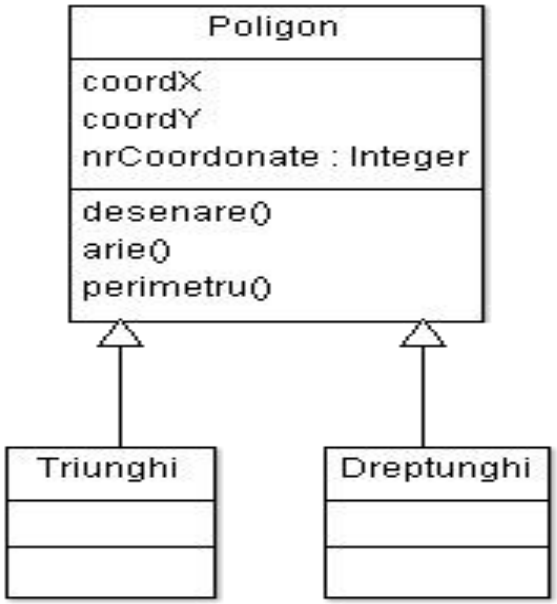
Poligon
coordX coordY nrCoordonate : Integer
desenare()

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

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

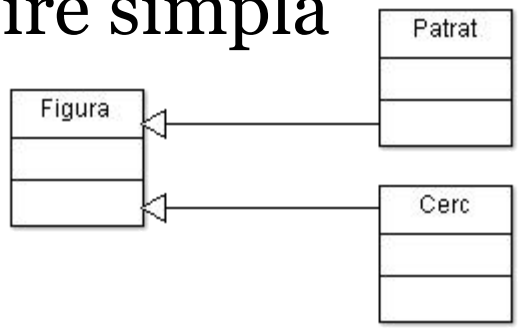
Exemplu – aplicare moștenire



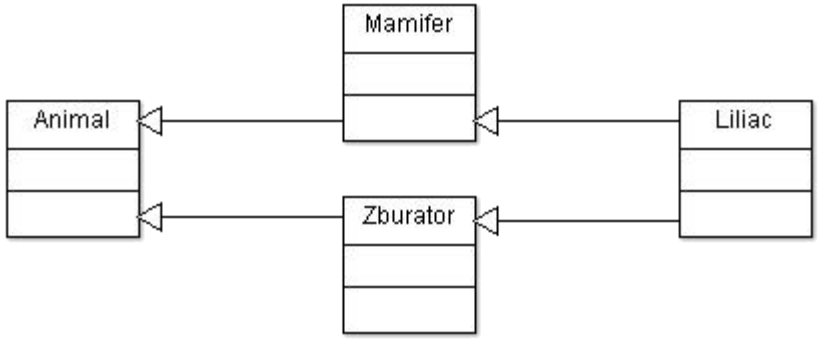


Tipuri de moștenire în C++

- Moștenire simplă



- Moștenire multiplă



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 access

- Funcţiile membre ale clasei derivate au acces la membrii publici şi protected ai clasei de bază
- Pentru a controla accesul la membrii clasei de bază sunt folosiţi specificatorii de acces

Clasa de Bază	Modificatorul de acces	Ce se poate accesa in 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 _a=a; //Ok
        int _b=b; //Ok
        int _c=c; //eroare: B::c este privat
    } };
class D2: protected B {
   void foo() {
        int _a=a; //Ok
        int _b=b; //Ok
        int _c=c; //eroare: B::c este privat
   } };
 class D3: public B {
     void foo() {
        int _a=a; //??
        int _b=b; //??
        int _c=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;
}
```

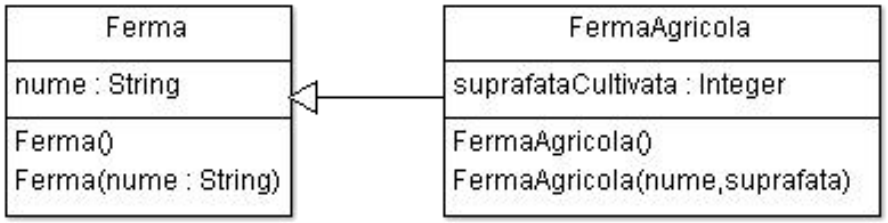
Ce se moşteneşte?

- În pricipiu, fiecare membru al clasei de bază este moştenit de clasa derivată
  - Doar cu diferite permisiuni de acces

- Dar, există câteva excepţii
  - Constructorii
  - Destructorii
  - Operatorul =
  - Funcţiile friend

Constructorii şi destructorii

- Constructori
  - Prima dată se apelează constructorul clasei de bază şi apoi constructorul clasei derivate
- Destructori
  - Sunt apelaţi în ordine inversă faţă de constructori, mai întâi se apelează destructorul clasei derivate apoi cel al clasei de bază



Constructorii și destructorii

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;
    }
};
```

Ferma Agricolă

```
class FermaAgricola : public Ferma{
  protected:
    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;
    }
};
```

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

Constructorii și destructorii

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;
  }
};
```

Ferma Agricolă

```
class FermaAgricola : public Ferma{
 protected:
  int suprafataCultivabila;
 public:
  FermaAgricola(){ cout << "FermaAgricola: Constructor implicit
     " << this <<< << endl;
  }
 FermaAgricola(char* nume, int suprafata):Ferma(nume){
     cout << " FermaAgricola : Constructor cu parametrii " << this
      <<endl;
     ...
  }
  ~ FermaAgricola(){
     cout << " FermaAgricola: Destructor " << this << endl;
  }
};
```

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

Constructorul de copiere

- Clasa derivată nu are definit un contructor de copiere => se apelează constructorul de copiere creat de compilator
- Clasa derivată are definit un contructor 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 şi 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: Destructor 0x28ff10 Ferma: Destructor 0x28ff10
```

Constructorul de copiere

- Apelarea constructorului supraclasei

```
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 supraclasei

```
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

- Funcţiile operator membre ale clasei de bază sunt moştenite de clasele derivate şi pot fi redefinite în clasele derivate
- EXCEPŢIE
  - 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 şi operatorul =() definit în clasa de bază

Conversii de tip

Permise

- D -> B
- *D -> *B
- *B -> (B*)*D

Nepermise

- B -> D
- *B -> *D

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

Redefinirea funcţiilor membre

- Ce este redefinirea funcţiilor?

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

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

```
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 '";
           afisare();
           cout <<"'\n"
       }
}
```

Output: Ferma agricola: 'fermaAgr cu suprafata 30' isi schimba numele in 'Ferma ABC 2’

Redefinirea funcţiilor membre

- Ce este redefinirea funcţiilor?

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

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

```
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"
       }
}
```

Output: Ferma agricola: 'fermaAgr cu suprafata 30' isi schimba numele in 'Ferma ABC 2’

Funcţii virtuale

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

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

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

Care este varianta de afişare dorită?

Ce se va afişa?

Output	
Varianta A	Varianta B
Ferma: Ferma Buzias	Ferma Agricola: Ferma Buzias cu suprafata 300

Funcţii virtuale

- Soluţia
  - Adăugarea unui câmp care specifică tipul obiectului
  - Funcţii viruale

- Tipuri de legături
  - Statică (early binding)
    - Versiunea funcţiei apelate se stabileşte în momentul compilării
  - Dinamică (late binding)
    - Versiunea funcţiei apelate se stabileşte în timpul execuţiei programului

Funcţii virtuale

- Sintaxă
  - virual prototipFuncţie;
- Caracteristici
  - Atributul virtual este moştenit
  - Este un tip special de funcţie care determină tipul derivat corespunzător pentru o funcţie cu acelasi prototip
  - Specificarea cuvântului „virtual” în faţa funcţiei
  - Sunt funcţii membre nestatice
  - Redefinirea şi redeclararea funcţiilor virtuale în clasele derivate nu este obligatorie
  - Constructori nu pot fi funcţii virtuale
  - Redefinirea sau schimbarea prototipului este acceptabilă doar dacă se modifică valoarea de return

Funcții virtuale

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

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

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

Output	
Varianta A	Varianta B
Ferma: Ferma Buzias	Ferma Agricola: Ferma Buzias cu suprafata 300

Destructori virtuali

- Ar trebui ca destructori claselor bază să fie declaraţi virtuali? De ce da sau de ce nu?

Destructori virtuali

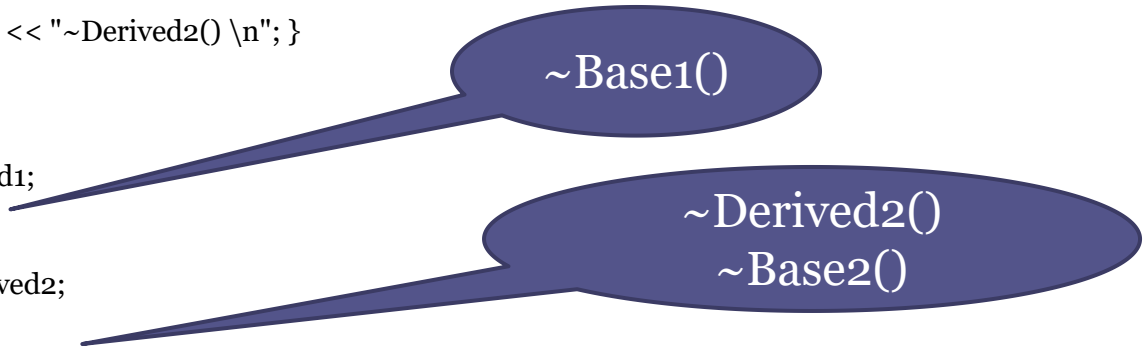
- Ar trebui ca destructori claselor bază să fie declaraţi virtuali? De ce da sau de ce nu?
  - Da! Trebuie să ştergem întotdeauna şi pointerii din subclase (altfel apare riscul de memory leaks)

Destructori virtuali

```
class Base1 {
  public:
    ~Base1() { std::cout << "~Base1() \n"; }
};
class Derived1 : public Base1 {
  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() {
    Base1* bp = new Derived1;
    delete bp;

     Base2* b2p = new Derived2;
     delete b2p;
}
```

Destructori virtuali

```
class Base1 {
  public:
    ~Base1() { std::cout << "~Base1() \n"; }
};
class Derived1 : public Base1 {
  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() {
    Base1* bp = new Derived1;
    delete bp;

     Base2* b2p = new Derived2;
     delete b2p;
}
```



Polimorfism

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

- Tipuri de polimorfism
  - Run-time
    - Moştenire şi funcţii virtuale
  - Compile-time
    - Template-uri
  - Ad-hoc
    - Supradefinirea operatorilor (operatorul + se comportă diferit pentru tipul int şi string)
  - Parametric
    - Conversia de tip (casting)

Posible erori

- Specificatorul de acces implicit pentru moştenire este private. O eroare des întâlnită este de a omite specificatorul de acces public

- Constructori nu pot fi funcţii virtuale

Sumar

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

Curs viitor

- Funcţii pur virtuale
- Clase abstracte
- Moştenirea multiplă
- Type casting