Polimorfismul. Funcţii Virtuale. Clase Abstracte

Mihai Gabroveanu

Ce este polimorfismul?

- **Polimorfismul** este capacitatea unor entități de a lua forme diferite.
- Etimologia *polimorm: Poly = multe + Morfm = forma*
- Este unul din conceptele esențiale din POO

Tipuri de polimorfism (I)

- *Polimorfismul parametric* – 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 binding*)).

Tipuri de polimorfism (II)

- *Polimorfismul de moștenire* – 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*)).
- În limbajul C++ este implementat cu ajutorul *funcțiilor virtuale*

Upcasting – Conversia Tipurilor

- **Upcasting** = conversia obiectelor claselor derivate către obiecte ale claselor de bază
- Considerăm următoarea ierahie de clase:

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

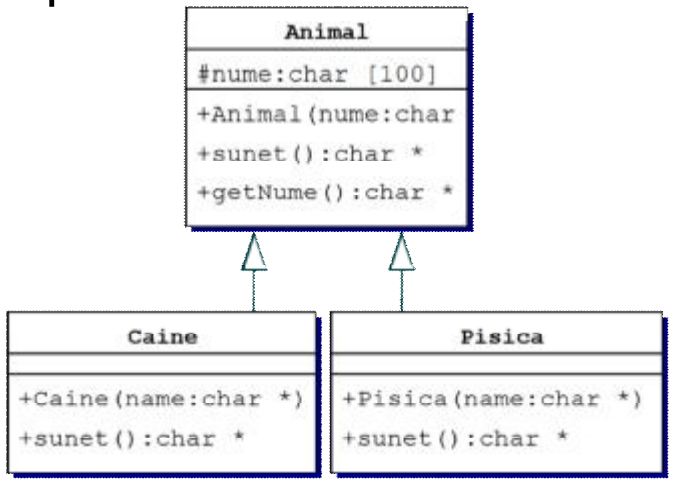
atunci:

```
B b, *pb; D d,*pd;
...
b = d;
pb=pd;
d=b
pd=pb;
```

Upcasing

Downcasting (incorect)

Exemplu



Exemplu (cont.)

```
class Animal {
  protected:
    char nume[100];
  public:
    Animal(char *nume){
     strcpy(this->nume,nume);
    }
   char* sunet(){
    return "nu stiu";
   }
   char* getNume(){
     return nume;
   }
};
```

```
class Pisica : public Animal{
  public:
  Pisica(char* nume):Animal(nume) {}
  char* sunet() {
    return "Miau!";
  }
};

class Caine : public Animal{
  public:
  Caine(char* nume):Animal(nume) {}
  char* sunet(){
    return "Ham!";
  }
};
```


Exemplu (cont.)

```
int main(){
  Animal *a = new Animal("Jerry");
  Pisica *t = new Pisica("Tom");
  Caine *c = new Caine("Azorel");

  cout << a->getNume() << ": " << a->sunet() << endl;
  cout << t->getNume() << ": " << t->sunet() << endl;
  cout << c->getNume() << ": " << c->sunet() << endl;
  a=t;
  cout << a->getNume() << ": " << a->sunet() << endl;
  c=a;//incorect
  c=t;//incorect
}

OUTPUT:
Jerry : nu stiu
Tom : Miau!
Azorel : Ham!
Tom : nu stiu! ???
```

Funcții virtuale

- O **funcție virtuală** este o funcție membră a unei clase care poate fi înlocuită, în momentul execuției programului, cu o funcție membru din clasa derivată.
- Sintaxa declarării:

```
class IdClasaBaza{
  virtual tip idMetodaV(lista_parametri);
};
class  IdClasaDerivata : IdClasaBaza{
  tip idMetodaV(lista_parametri);
};
```


Apelul funcțiilor virtuale

- Cuvântul cheie **virtual** permite implementarea *legăturilor dinamice* la apelul funcției prin intermediul unui pointer al clasei de bază. Mai exact, dacă un pointer al clasei de bază indică către un obiect din ierarhie atunci se selectează la rulare versiunea de funcție corespunzătoare tipului de obiect referit.
- De exemplu:

```
IdClasaDerivata d;
IdClasaBaza *p = &d;
p->idMetodaV();  //va apela metoda definită în clasa IdClasaDerivata.
IdClasaBaza b=d;
b.idMetodaV();  //va apela metoda definită în clasa IdClasaBaza
```


Exemplu

```
class Animal {
  protected:
    char nume[100];
  public:
    Animal(char *nume);
    virtual char* sunet();
    char* getNume();
};

void main(){
  Animal* animale[] =  {
    new Pisica("Felix"),
    new Caine("Azorel"),
    new Pisica("Tom")
  };
  for(int i = 0; i < 3; i++)  {
    cout<<animale[i]->getNume()<< ":";
    cout<<animale[i]->sunet()<< endl;
  }
}
```

sunet() devine virtuală

```
Output:
Felix: Miau!
Azorel: Ham!
Tom: Miau!
```


Avantaje

- Putem trata colecții de obiecte ale aceleiași ierarhii într-o manieră unitară deoarece funcţia apelată virtual este identificată la rulare cu funcţia membră din clasa căreia îi aparţine obiectul
- Putem asigura independența implementărilor

Constructori și Destructori

- Constructori **nu** pot fi funcții virtuale.
- Destructori **pot** fi funcții virtuale. Uneori chiar este necesar acest lucru.

Exemplu

```
class Persoana {
  protected:
    char *nume;
  public:
     Persoana(char *nume="") {
     this->nume = new char[strlen(nume)+1];
     strcpy(this->nume, nume);
  }
    virtual ~Persoana() {
    if (nume){
      delete []nume;
      nume = 0;
    }
    cout << "Distruge obiectul de Persoana " <<
endl;
  }
 };

 class Angajat : public Persoana {
    char * functia;
 public:
    Angajat(char *nume="", char * functia):
Persoana(nume) {
    this->functia = new char[strlen(functia)+1];
    strcpy(this->functia, functia);
    }
    ~Angajat() {
    if (functia){
      delete []functia;
      functia = 0;
    }
    cout << "Distruge obiectul Angajat " << endl;
    }
 };

 int main() {
    Persoana* p = new Angajat("Mircea","Sudor");
    delete p; //Apelează destructorii ~Persoana()
si ~Angajat()

    return 0;
 }
```

Ce se întâmplă dacă destructorul clasei persoana nu este virtual?

Funcții virtuale pure

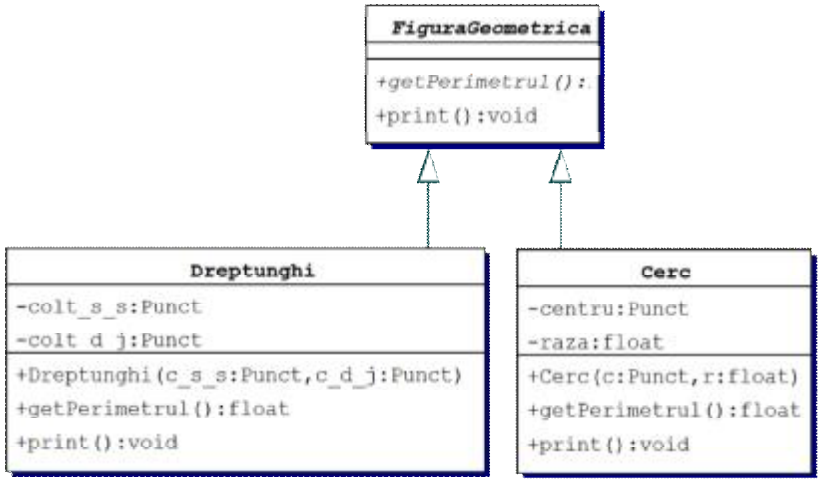
- **Funcții virtuale pure** sunt funcții virtuale ce sunt doar declarate în clasa de bază, urmând ca acestea sa fie definite în clasele derivate.
- Sintaxa:

```
class IdClasaBaza{
   virtual tip idMetodaVirtPura(lista_parametri) = 0;
};
```


Clase Abstracte

- O clasă abstractă este o clasă care conține cel puțin o funcție virtuală pură.
- Nu putem crea instanțe ale claselor abstracte
- Puntem însă declara pointeri
- Dacă o clasă moștenește o clasă abstractă și nu implementează toate metodele virtuale pure atunci ea devine clasă abstractă.

Exemplu



Exemplu (cont.)

Clasa abstracta

Metoda virtuala pură
(nu are definiție)

Metoda e redefinită

```
class FiguraGeometrica {
      public:
         virtual float getPerimetrul() = 0;
          virtual void print();

};

void FiguraGeometrica::print(){
     cout<<"Figura Geometrica Oarecare"<<endl;
}

class Cerc: public FiguraGeometrica {
        Punct centru;
        float raza;
      public:
          Cerc (Punct c, float r);
          float getPerimetrul();
          void print();
};
Cerc::Cerc(Punct c, float r):centru(c),
                                raza(r){
}

float Cerc::getPerimetrul() {
   return 2*M_PI*raza;
}
void Cerc::print(){
   cout<<"Cerc ("<<centru<<","<<raza<<")"<<endl;
}
```


Exemplu (cont.)

```
class Dreptunghi: public FiguraGeometrica {
      Punct colt_s_s; //colt stanga sus
      Punct colt_d_j; //colt dreapta jos
     public:
      Dreptunghi (Punct c_s_s, Punct c_d_j);
      float getPerimetrul();
      void print();
};

Dreptunghi::Dreptunghi(Punct c_s_s, Punct
c_d_j): colt_s_s(c_s_s), colt_d_j(c_d_j){
}
float Dreptunghi::getPerimetrul() {
      return 2*(colt_d_j.getX()-
colt_s_s.getX())+2*(colt_s_s.getY()-
colt_d_j.getY());
}
```

```
void Dreptunghi::print(){
     cout<<"Dreptunghi
["<<colt_s_s<<","<<colt_d_j<<"]"<<endl;;
}

void main(){
 FiguraGeometrica *f[]={new Cerc(Punct(0,0),1),
       new Dreptunghi(Punct(0,1),Punct(2,0))};
 for(int i=0;i<2;i++){
   f[i]->print();
   cout<<"Perimetrul="<<f[i]->
getPerimetrul()<<endl;
    }
}
```


Temă

- Implementati următoarea ierahie de clase: Animal, Mamifer, AnimalZburator, Liliac.
- Imlementați o ierarhie de clase care reprezintă articolele dintr-o bibliotecă.