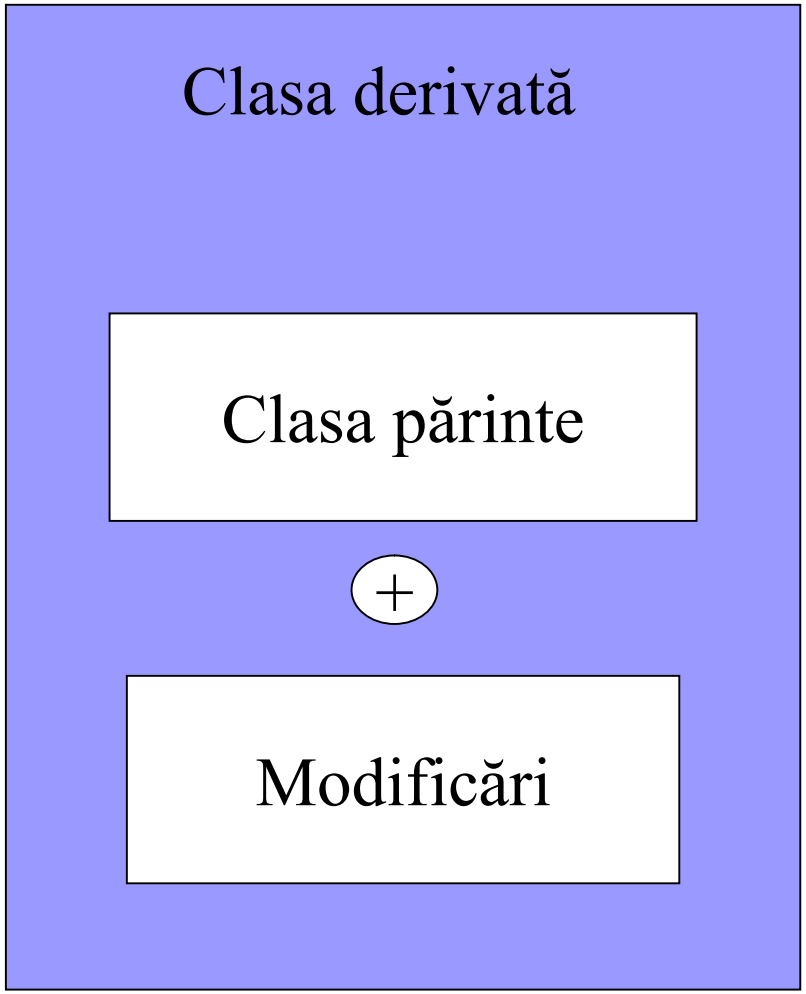
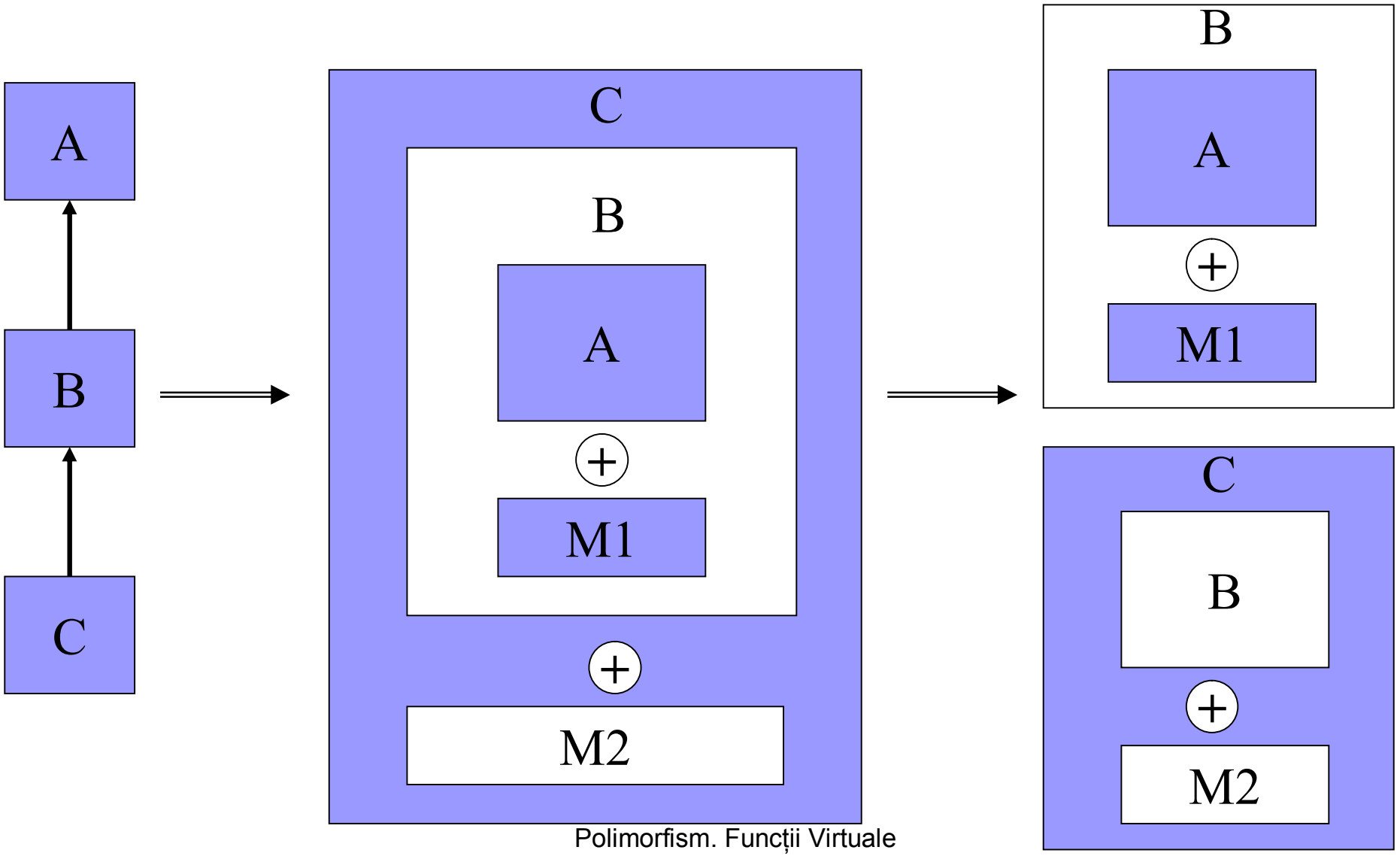
Polimorfism. Funcții Virtuale. Clase Abstracte

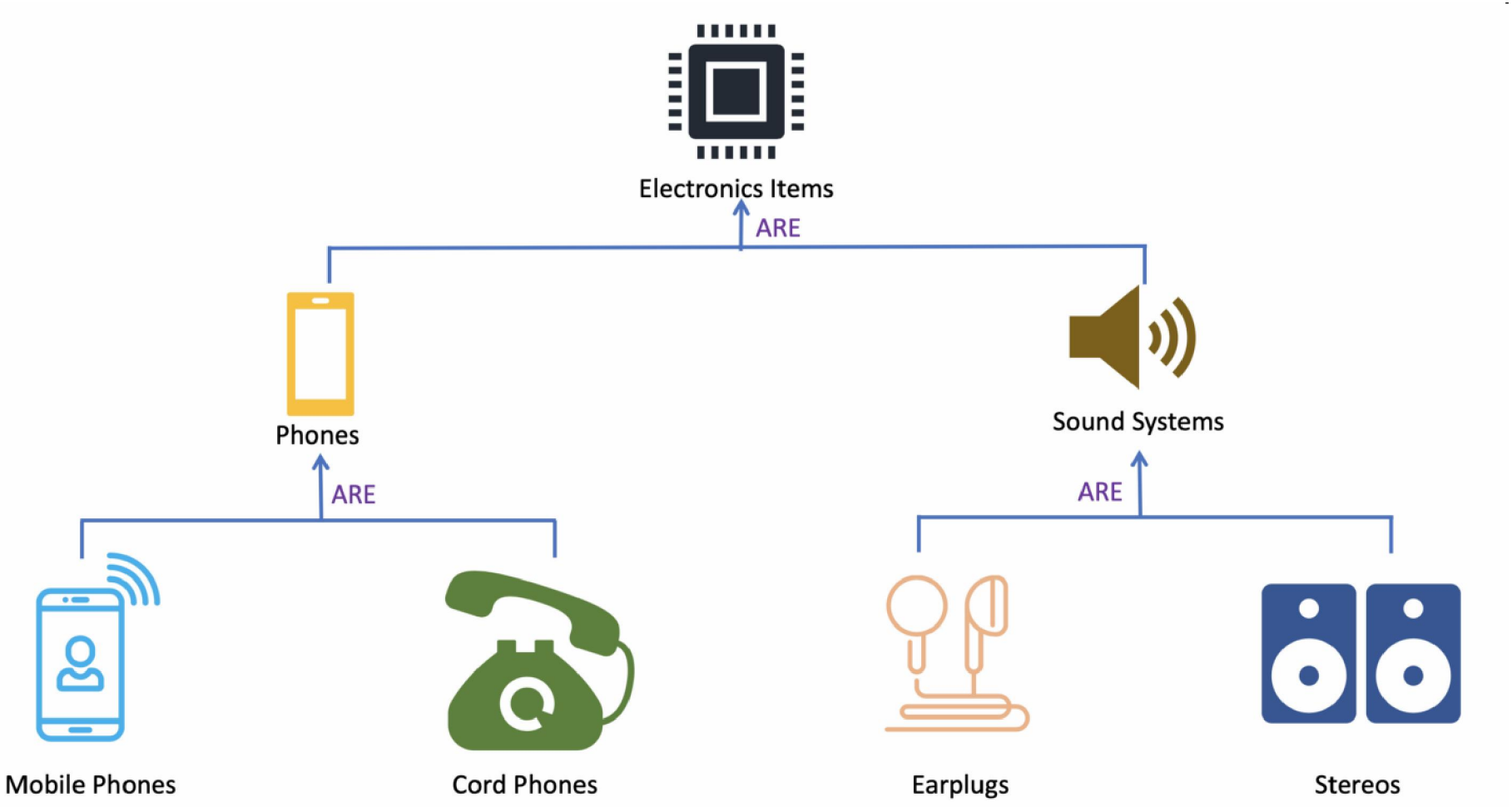
Moștenire



Moștenire



lerarhie de clase



Moştenire

- Moștenirea permite *definirea de clase noi* (clase derivate) reutilizând clase existente (clase de bază).
- Clasa nou creată *moștenește* comportamentul (metode) și caracteristicile (atributele membre) de la clasa de bază.
- Dacă `A` și `B` sunt două clase unde `B` moștenește de la clasa `A` (`B` este derivată din clasa `A` sau clasa `B` este o specializare a clasei `A`) atunci:
  - clasa `B` are toate metodele şi variabilele membre din clasa `A`;
  - clasa `B` poate redefini metode din clasa `A`;
  - clasa `B` poate adăuga noi membri (variabile, metode) pe lângă cele moștenite de la clasa `A`.

Moştenire

- Dacă clasa `B` moștenește de la clasa `A` atunci:
  - orice obiect de tip `B` are toate atributele (variabilele membre) din clasa `A`;
  - Metodele (funcțiile) din clasa `A` pot fi aplicate şi asupra obiectelor de tip `B` (dacă vizibilitatea permite);
  - clasa `B` poate adăuga atribute membre și/sau metode pe lângă cele moștenite din `A`.
- *membrii moșteniți* (metode, atribute) sunt membrii definiți în clasa A și nemodificați în clasa B.
- *membrii redefiniți* (*overridden*) sunt definiți în A și în B (în B se crează o nouă definiție).
- *membrii adăugați* sunt definiți doar în B.

Moştenire

- **Supraîncărcare** (**overloading**)
  - acelaşi nume, parametri diferiți, acelaşi context sau context diferit (clasă sau global).
- **Redefinire** (**overriding**)
  - acelaşi nume, aceiaşi parametri, clase diferite (dar din *aceeaşi ierarhie*).

Ce este polimorfismul?

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

Tipuri de polimorfism (I)

- *Polimorfism parametric* – mecanismul prin care putem defini *mai multe metode cu acelaşi nume în aceeaşi clasă*, funcții ce trebuie să difere prin numărul şi / sau tipul parametrilor.
- Selectarea funcției ce se va apela se realizează la compilare - legarea timpurie (*early binding*).

Tipuri de polimorfism (II)

- *Polimorfism de moştenire* – mecanismul prin care o metodă din clasa de bază este **redefinită** cu aceiaşi parametri în clasele derivate.
- Selecția funcției ce se va apela, se va realiza în momentul rulării aplicației - legarea întârziată (*late binding, dynamic binding, runtime binding*).
- În limbajul C++ acest mecanism este implementat cu ajutorul *funcțiilor virtuale.*

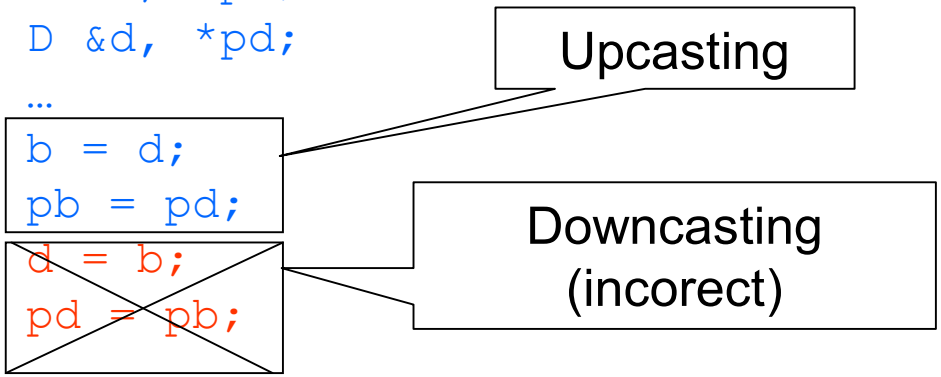
Upcasting – Conversia Tipurilor

- Între obiecte
  - obiect *clasă Derivată* -> *obiect clasa Bază*: conversie cu trunchiere;
  - obiect *clasă Bază* -> *obiect clasă Derivată*: nu este permis.
- *Upcasting* - conversie implicită pointer/referință *clasă Derivată* -> *clasă Bază.*
- *Downcasting* - conversie explicită (prin operatorul de conversie) pointer/referință *clasă Bază* -> *clasă Derivată*.
- Să considerăm următoarea ierarhie de clase:

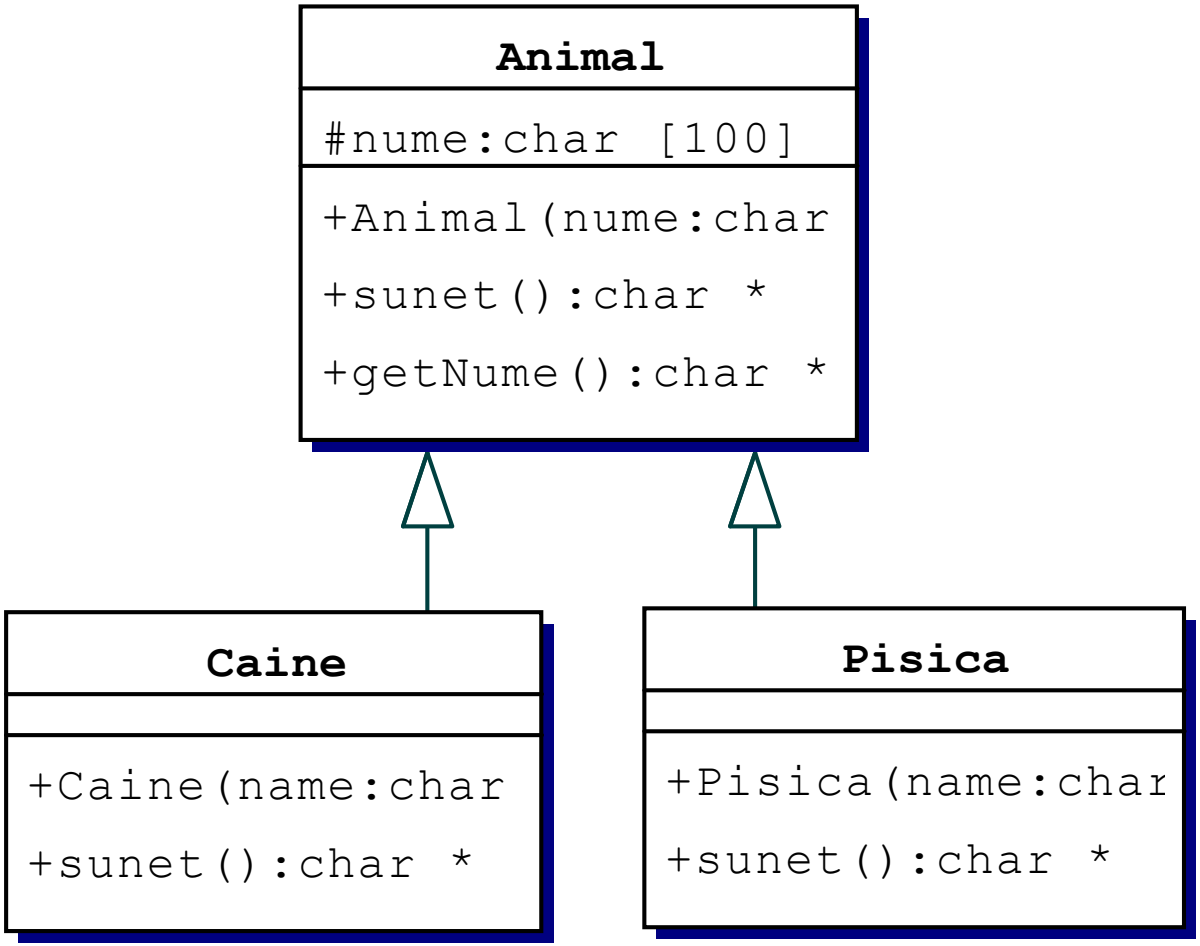
```
class B;
class D: public B { };
```

atunci:

```
B &b, *pb;
D &d, *pd;
…
b = d;
pb = pd;
d = b;
pd = pb;
```



Exemplu


Animal
#nume:char [100]
+Animal(nume:char
+sunet():char *
+getNume():char *
Caine
+Caine(name:char
+sunet():char *
Pisica
+Pisica(name:char
+sunet():char *

Exemplu

```
class Animal {
  protected:
    char nume[100];
  public:
    Animal(const char *nume) {
      strcpy(this->nume, nume);
    }

    const char* sunet() {
      return "nu stiu";
    }

    char* getNume() {
      return nume;
    }
};
```

```
class Pisica : public Animal {
  public:
    Pisica(const char* nume):Animal(nume) {}

    const char* sunet() {
      return "Miau!";
    }
};

class Caine : public Animal {
  public:
    Caine(const char* nume):Animal(nume) {}

    const 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
  return 0;
}

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

Funcții virtuale

- Mecanism ce permite referirea corectă a funcțiilor dintr-o *clasă derivată* prin intermediul unui pointer sau referință la o *clasă de bază*, inițializat cu adresa unui obiect din *clasa derivată*.
- Sintaxa declarării unei funcții virtuale:

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

Funcții virtuale

- Atributul `virtual` se moşteneşte.
- O funcție devine virtuală numai odată cu specificarea cuvântului-cheie „`virtual`” la începutul antetului funcției.
- Funcțiile virtuale pot fi numai funcții membre *nestatice*.
- Redefinirea şi redeclararea funcțiilor virtuale în clasele derivate nu este obligatorie.
- Constructorii nu pot fi funcții virtuale.
- Redefinirea sau schimbarea prototipului este acceptabilă doar dacă se modifică valoarea de *return*.

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 ierahie, 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

sunet() devine virtuală

```
class Animal {
  protected:
    char nume[100];

  public:
    Animal(const char* nume);

    virtual const char* sunet();
    char* getNume();
};
…
int 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;
  }
  return 0;
}
```

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

Avantaje

- Putem trata colecții de obiecte ale aceleiaşi ierarhii într-o manieră unitară, deoarece funcția apelată virtual este identificată la rulare cu funcția membră din clasa căreia îi aparține obiectul.
- Putem asigura independența implementărilor.

Polimorfism

- Principiul substituției (Liskov)
  - Funcțiile ce utilizează pointeri sau referințe către instanțe ale claselor de bază trebuie să poată folosi instanțe ale claselor derivate fără să își dea seama de acest lucru.
- sau
  - Un obiect de tipul clasei derivate se poate folosi în orice loc (context) unde se cere un obiect de tipul clasei de bază.

Upcasting/Downcasting (I)

- **Upcast** - conversia unui pointer sau referință având ca tip o *clasă derivată* către un pointer sau referință având ca tip o *clasă de bază* (mergând în sus în arborele de moştenire).
  - ajută la realizarea conceptului de interfață în C++.
  - când o funcție este apelată de un pointer sau referință având ca tip o *clasă de bază* (ce indică sau se referă la o parte dintr-o *clasă derivată a sa*), atunci va fi invocată funcția membru corectă a acelei *clase derivate*.

Upcasting/Downcasting (II)

- **Downcast** - conversia unui pointer sau referință având ca tip o *clasă de bază* la un pointer sau referință având ca tip o *clasă derivată* (mergând în jos în arborele de moştenire).
  - În timpul rulării unei aplicații acest lucru este realizat în condiții de siguranță cu ajutorul operatorului `dynamic_cast`.

Constructori şi Destructori

- Constructorii nu pot fi funcții virtuale.
- Destructorii pot fi funcții virtuale. Uneori chiar este necesar acest lucru.

Exemplu

```
class Persoana {
  protected:
    char *nume;

  public:
    Persoana(const 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 tip Persoana"
           << endl;
    }
};

class Angajat : public Persoana {
  private:
    char *functia;

  public:
    Angajat(const char* nume = "",
            const 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 ~Angajat()
si ~Persoana()

  return 0;
}
```

Ce se întâmplă dacă destructorul clasei Persoana nu este virtual?

Funcții virtuale pure

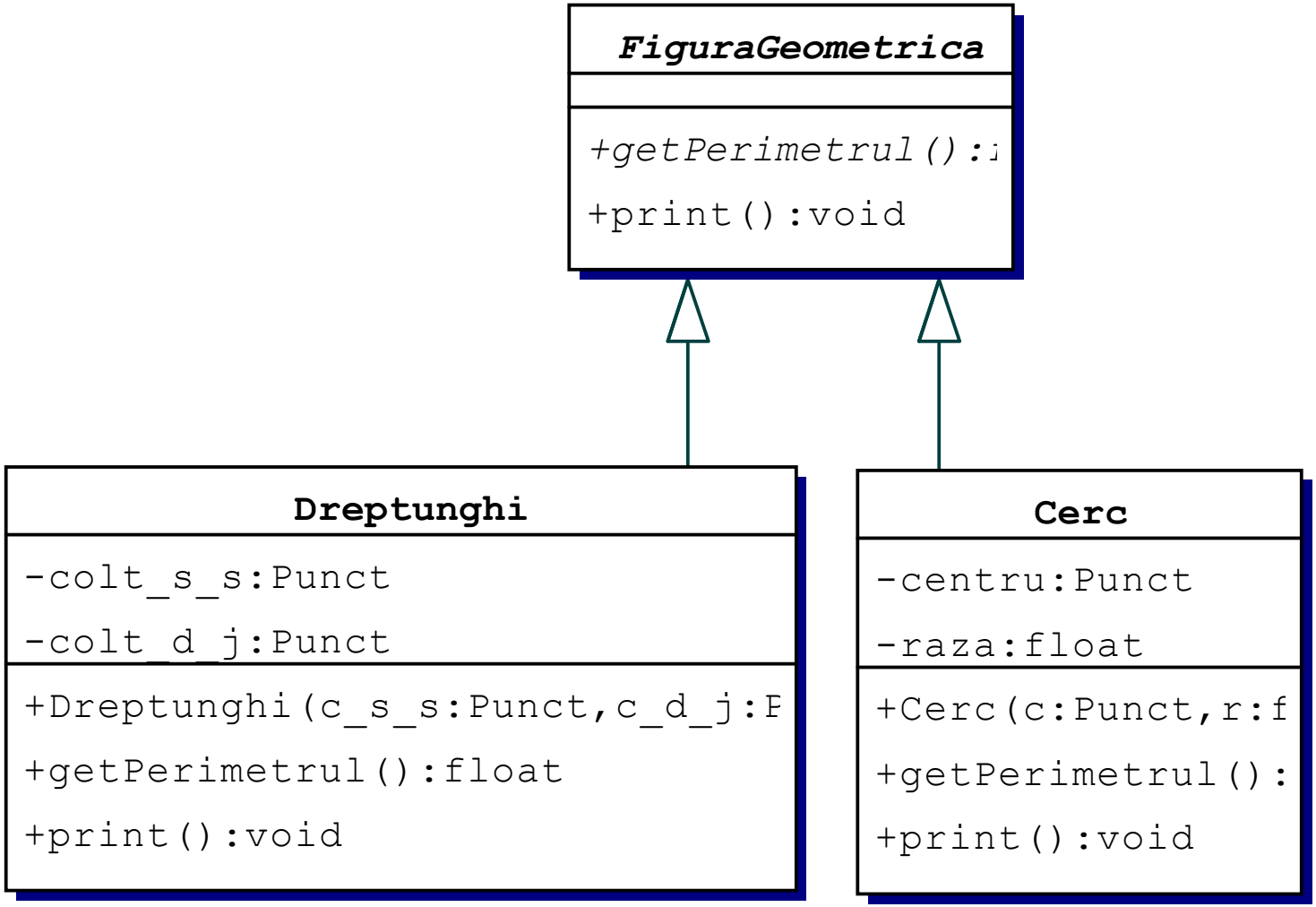
- **Funcții virtuale pure** - sunt funcții virtuale ce sunt doar declarate în clasa de bază, urmând ca acestea să fie definite în clasele derivate.
- Sintaxa:

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

Clase Abstracte

- O *clasă abstractă* este o clasă ce conține cel puțin o *funcție virtuală pură*.
- Nu putem crea instanțe ale claselor abstracte.
- Se pot însă declara pointeri către clase abstracte.
- Dacă o clasă moşteneşte o clasă abstractă şi nu implementează toate metodele virtuale pure, atunci ea devine clasă abstractă.

Exemplul 1


FiguraGeometrica
+getPerimetrul():
+print():void
Dreptunghi
-colt_s_s:Punct
-colt_d_j:Punct
+Dreptunghi(c_s_s:Punct,c_d_j:P
+getPerimetrul():float
+print():void
Cerc
-centru:Punct
-raza:float
+Cerc(c:Punct,r:f
+getPerimetrul():
+print():void

Exemplul 1 (cont.)

Clasă abstractă

Metodă virtuală pură
(nu are definiție)

Metoda este redefinită

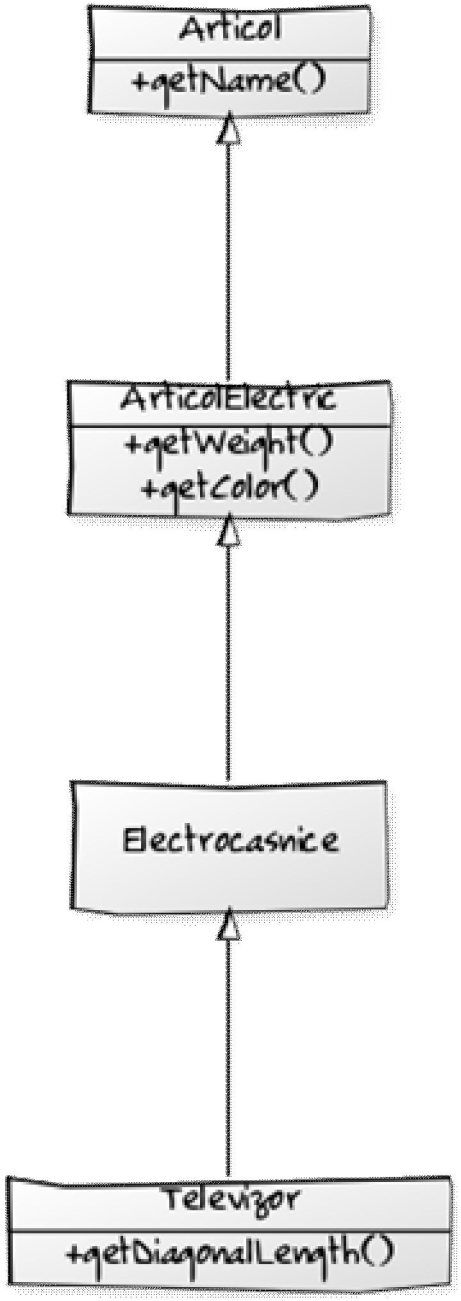
```
class FiguraGeometrica {
  public:
    virtual float getPerimetrul() = 0;
    virtual void print();

};

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

class Cerc: public FiguraGeometrica {
  private:
    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;
}
```

Exemplul 1 (cont.)

```
class Dreptunghi: public FiguraGeometrica {
    Punct colt_ss; //colt stanga sus
    Punct colt_dj; //colt dreapta jos
  public:
    Dreptunghi (Punct c_ss, Punct c_dj);

    float getPerimetrul();
    void print();
};

Dreptunghi::Dreptunghi(Punct css, Punct cdj):
                    colt_ss(css), colt_dj(cdj) {
}

float Dreptunghi::getPerimetrul() {
  return 2*(colt_dj.getX() - colt_ss.getX()) +
         2*(colt_ss.getY() - colt_dj.getY());
}
```

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

int 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;
 }
 return 0;
}
```

Exemplul 2


Articol
+getName()
ArticolElectric
+getWeight()
+getColor()
Electrocasnice
Televizor
+getDiagonalLength()

Exemplul 2 (cont.)

```
class Articol {
  public:
    void getName() {
      cout << "Articol" << endl;
    }
};

class ArticolElectric: public Articol {
  public:
    void getWeight() {
      cout << "unknown" << endl;
    }

    void getColor() {
      cout << "Gray" << endl;
    }
};

class Electrocasnic: public ArticolElectric {
  public:
    void getName() {
      cout << "Articol electrocasnic" << endl;
    }
    void getColor() {
      cout << "White" << endl;
    }
};

class Televizor: public Electrocasnic {
  public:
    void getName() {
      cout << "Televizor" << endl;
    }

    void getWeight() {
      cout << "8 kg" << endl;
    }

    void getDiagonalLength() {
      cout << "DiagonalLength" << endl;
    }
};
```

Exemplul 2 (cont.)

```
int main() {
  Articol a;
  ArticolElectric ae;
  Electrocasnic ec;
  Televizor tv;

  Articol* pa = &a;
  ArticolElectric* pae = &ae;
  Electrocasnic* pec = &ec;
  Televizor* ptv = &tv;

  ((Articol*)pec)->getName();
  ((ArticolElectric*)pec)->getName();
  ((Electrocasnic*)pec)->getName();
  ((Televizor*)pec)->getName();
  cout << "\n";
  ((ArticolElectric*)pec)->getWeight();
  cout << "\n";
  ptv = (Televizor*)&ec;
  ((ArticolElectric*)ptv)->getColor();
  ((Electrocasnic*)ptv)->getColor();
  ((Televizor*)ptv)->getColor();

  return 0;
}
```

```
Output:
  Articol
  Articol
  Articol electrocasnic
  Televizor

  Unknown

  Gray
  White
  White
```

Exemplul 2

Articol	ArticolElectric	Electrocasnic	Televizor
getName()	-	getName()	getName()
-	getWeight()	-	getWeight()
-	getColor()	getColor()	-
-	-	-	getDiagonalLength()

Exemplul 2 – virtual

```
class Articol {
  public:
    virtual void getName() {
      cout << "Articol" << endl;
    }
};

class ArticolElectric: public Articol {
  public:
    virtual void getWeight() {
      cout << "unknown" << endl;
    }

    virtual void getColor() {
      cout << "Gray" << endl;
    }
};

class Electrocasnic: public ArticolElectric {
  public:
    void getName() {
      cout << "Articol electrocasnic" << endl;
    }
    void getColor() {
      cout << "White" << endl;
    }
};

class Televizor: public Electrocasnic {
  public:
    void getName() {
      cout << "Televizor" << endl;
    }

    void getWeight() {
      cout << "8 kg" << endl;
    }

    virtual void getDiagonalLength() {
      cout << "DiagonalLength" << endl;
    }
};
```

Exemplul 2 – virtual (cont.)

```
int main() {
  Articol a;
  ArticolElectric ae;
  Electrocasnic ec;
  Televizor tv;

  Articol* pa = &a;
  ArticolElectric* pae = &ae;
  Electrocasnic* pec = &ec;
  Televizor* ptv = &tv;

  ((Articol*)pec)->getName();
  ((ArticolElectric*)pec)->getName();
  ((Electrocasnic*)pec)->getName();
  ((Televizor*)pec)->getName();
  cout << "\n";
  ((ArticolElectric*)pec)->getWeight();
  cout << "\n";
  ptv = (Televizor*)&ec;
  ((ArticolElectric*)ptv)->getColor();
  ((Electrocasnic*)ptv)->getColor();
  ((Televizor*)ptv)->getColor();

  return 0;
}
```

```
Output:
  Articol electrocasnic
  Articol electrocasnic
  Articol electrocasnic
  Articol electrocasnic
  Unknown

  White
  White
  White
```

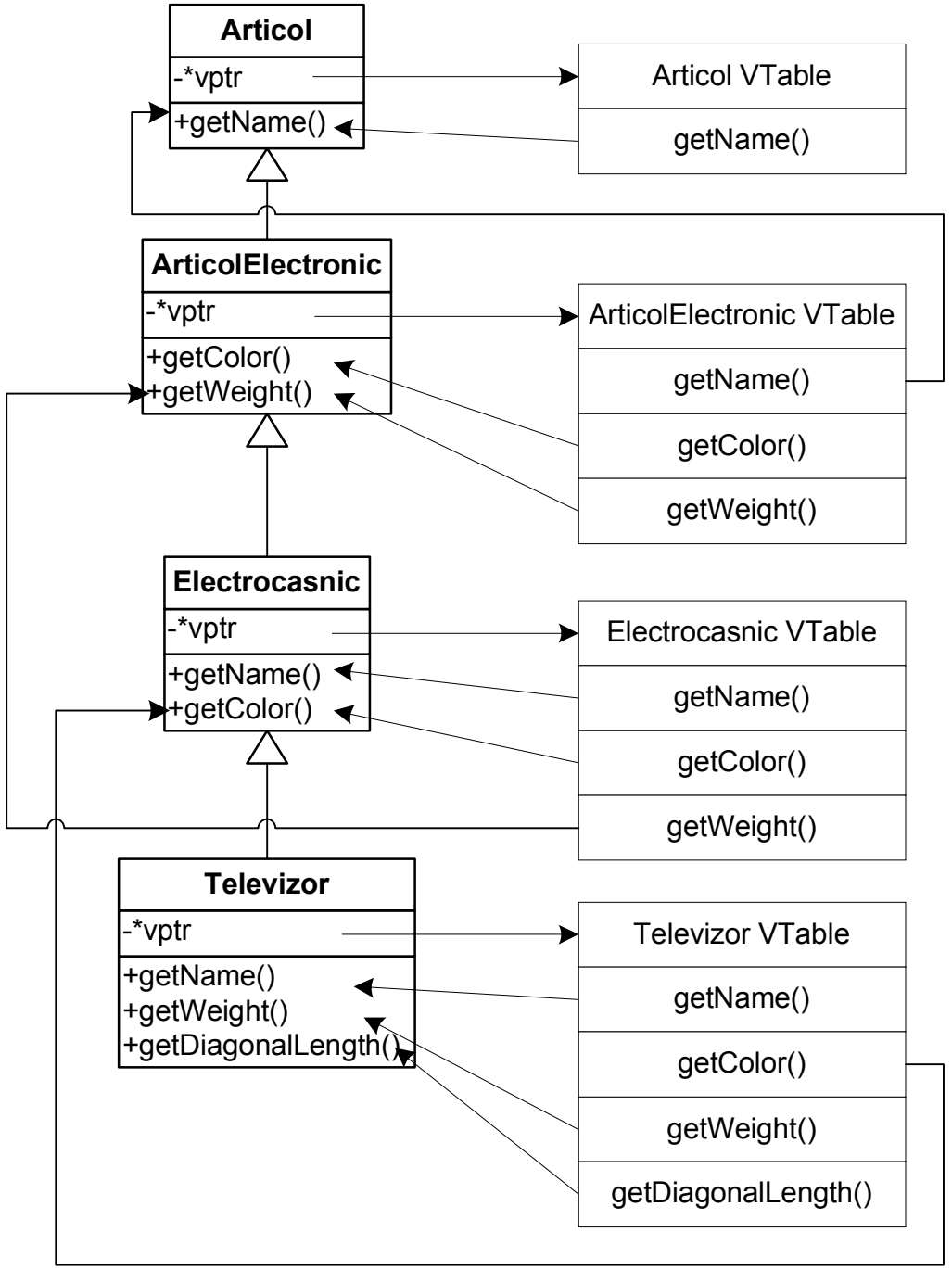
Exemplul 2 – cu şi fără virtual

Codul sursa	Output exemplul 2	Output exemplul 2 + virtual
((Articol*)pec)->getName();	Articol	Articol electrocasnic
((ArticolElectric*)pec)->getName();	Articol	Articol electrocasnic
((Electrocasnic*)pec)->getName();	Articol electrocasnic	Articol electrocasnic
((Televizor*)pec)->getName();	Televizor	Articol electrocasnic
((ArticolElectric*)pec)->getWeight();	unknown	unknown
((ArticolElectric*)ptv)->getColor();	Gray	White
((Electrocasnic*)ptv)->getColor();	White	White
((Televizor*)ptv)->getColor();	White	White

Funcții virtuale

- Tabela de funcții virtuale (**v-table**, **vtbl**)
  - este creată la nivelul fiecărei clase ce conține cel puțin o funcție membră virtuală;
  - conține pointeri către funcțiile membru virtuale;
  - este actualizată cu adresele funcțiilor redefinite pentru fiecare clasă nouă din ierarhie.
- Pointer la tabela de funcții virtuale (**v-pointer**, **vptr**)
  - tabela de funcții virtuale este referită printr-un pointer;
  - dată membru ascunsă pentru fiecare obiect dintr-o clasă ce conține cel puțin o funcție membru virtuală;
  - este adăugat de compilator şi inițializat în constructorul clasei.

Exemplul 2


Articol
-*vptr
+getName()
Articol VTable
getName()
ArticolElectronic
-*vptr
+getColor()
+getWeight()
ArticolElectronic VTable
getName()
getColor()
getWeight()
Electrocasnic
-*vptr
+getName()
+getColor()
Electrocasnic VTable
getName()
getColor()
getWeight()
Televizor
-*vptr
+getName()
+getWeight()
+getDiagonalLength()
Televizor VTable
getName()
getColor()
getWeight()
getDiagonalLength()

Temă

- Implementați o ierarhie de clase ce reprezintă articolele dintr-o bibliotecă.