

Capitolul 3. Moștenire

Concepte de bază

Concepte de bază

def

Moștenire = procesul prin care o clasă (de bază) este extinsă într-o nouă clasă (derivată) prin preluarea datelor și funcțiilor membre.

- Clasa derivată “moștenește” caracteristicile clasei de bază.
- Clasa derivată poate adăuga informații suplimentare la cele moștenite.
- Definește o relație de tipul “este o/un” (is a) => ierarhizare.
- Se previne rescrierea codului atunci când se extinde funcționalitatea unei clase.
- Clasa de bază trebuie definită complet înainte de clasa derivată.

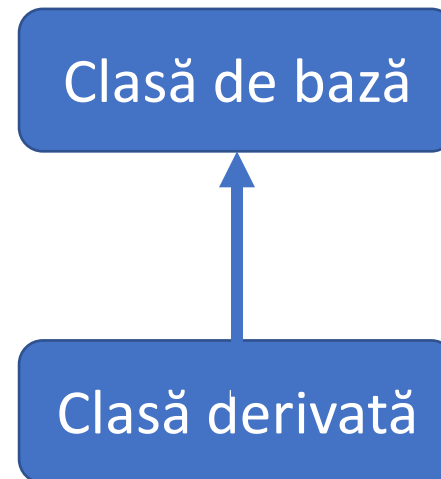
Principiile POO

1. Încapsulare = gruparea/învelirea/încapsularea datelor și a funcțiilor ce acționează asupra acestora într-un singur container (clasă).
2. Abstractizare = procedeul prin care se expun lumii exterioare doar funcționalitățile importante, fără a se intra în prea multe detalii.
3. Moștenire = procesul prin care o clasă este extinsă într-o nouă clasă prin preluarea datelor și funcțiilor membre.
4. Polimorfism

Concepte de bază

Sintaxă:

```
class <nume_derivata>:  
    [virtual] [specificator_acces] <nume_baza_1>,  
    [virtual] [specificator_acces] <nume_baza_2>, ... {  
    ...  
};
```



Concepte de bază

Accesul oferit de specificatorii de acces din clasă

	Specificator acces		
	public	protected	private
Membrii aceleiași clase	Da	Da	Da
Membrii clasei derivate	Da	Da	Nu
Non-membri (exterior)	Da	Nu	Nu

Specificatorii de acces rezultați în clasa derivată

Specificator acces în clasa de bază	Specificator acces moștenire		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	inaccesibil	inaccesibil	inaccesibil

← Dacă nu se declară explicit un specificator (default)

Exemplu moștenire public

```
#include <iostream>

class Animal{
private:
    int varsta;
protected:
    std::string nume;
public:
    float greutate;
};

class Caine : public Animal{
private:
    std::string talie;
protected:
    bool blana_lunga;
public:
    int lungime_coadă;

    void afisare_caine(){
        std::cout << varsta << std::endl; // eroare: varsta este private in baza
        std::cout << nume << std::endl; // OK: nume este protected in baza => protected in derivata
        std::cout << greutate << std::endl; // OK: greutate este public in baza => public in derivata
    }
};

int main(){
    Animal a;
    a.varsta = 5; // eroare: varsta este private in baza
    a.nume = "Animal X"; // eroare: nume este protected in baza
    a.greutate = 3.2 // OK: greutate este public in baza

    Caine c;
    c.varsta = 3; // eroare: varsta nu este accesibil in derivata
    c.nume = "Caine Y"; // eroare: nume este protected in derivata
    c.greutate = 4.7; // OK: greutate este public in derivata
    c.talie = "mare"; // eroare: talie este private in derivata
    c.blana_lunga = true; // eroare: blana_lunga este protected in
    derivata
    c.lungime_coadă = 23; // OK: lungime_coadă este public in derivata

    return 0;
}
```

Exemplu moștenire protected

```
#include <iostream>

class Animal{
private:
    int varsta;
protected:
    std::string nume;
public:
    float greutate;
};

class Caine : protected Animal{
private:
    std::string talie;
protected:
    bool blana_lunga;
public:
    int lungime_coadă;

    void afisare_caine(){
        std::cout << varsta << std::endl; // eroare: varsta este private in baza
        std::cout << nume << std::endl; // OK: nume este protected in baza => protected in derivata
        std::cout << greutate << std::endl; // OK: greutate este public in baza => protected in derivata
    }
};

int main(){
    Animal a;
    a.varsta = 5; // eroare: varsta este private in baza
    a.nume = "Animal X"; // eroare: nume este protected in baza
    a.greutate = 3.2 // OK: greutate este public in baza

    Caine c;
    c.varsta = 3; // eroare: varsta nu este accesibil in derivata
    c.nume = "Caine Y"; // eroare: nume este protected in derivata
    c.greutate = 4.7; // eroare: greutate este protected in derivata
    c.talie = "mare"; // eroare: talie este private in derivata
    c.blana_lunga = true; // eroare: blana_lunga este protected in
    derivata
    c.lungime_coadă = 23; // OK: lungime_coadă este public in derivata

    return 0;
}
```


Exemplu moștenire private

```
#include <iostream>

class Animal{
private:
    int varsta;
protected:
    std::string nume;
public:
    float greutate;
};

class Caine : private Animal{
private:
    std::string talie;
protected:
    bool blana_lunga;
public:
    int lungime_coadă;


    void afisare_caine(){
        std::cout << varsta << std::endl; // eroare: varsta este private in baza
        std::cout << nume << std::endl; // OK: nume este protected in baza => private in derivata
        std::cout << greutate << std::endl; // OK: greutate este public in baza => private in derivata
    }
};

int main(){
    Animal a;
    a.varsta = 5; // eroare: varsta este private in baza
    a.nume = "Animal X"; // eroare: nume este protected in baza
    a.greutate = 3.2 // OK: greutate este public in baza

    Caine c;
    c.varsta = 3; // eroare: varsta nu este accesibil in derivata
    c.nume = "Caine Y"; // eroare: nume este private in derivata
    c.greutate = 4.7; // eroare: greutate este private in derivata
    c.talie = "mare"; // eroare: talie este private in derivata
    c.blana_lunga = true; // eroare: blana_lunga este protected in
    derivata
    c.lungime_coadă = 23; // OK: lungime_coadă este public in derivata

    return 0;
}
```

Concepte de bază

 O clasă derivată poate modifica accesul datelor membre pe care le moștenește din clasa de bază prin plasarea calificării complete a datei membre în noua zonă de acces.

Exemplu moștenire public

```
#include <iostream>

class Animal{
private:
    int varsta;
protected:
    std::string nume;
public:
    float greutate;
};

class Caine : private Animal{
private:
    std::string talie;
protected:
    bool blana_lunga;
public:
    int lungime_coadă;
    Animal::greutate;

    void afisare_caine(){
        std::cout << varsta << std::endl; // eroare: varsta este private in baza
        std::cout << nume << std::endl; // OK: nume este protected in baza => private in derivata
        std::cout << greutate << std::endl; // OK: greutate este public in baza => accesibil in derivata
        => modificat explicit in public
    }
};

int main(){
    Animal a;
    a.varsta = 5; // eroare: varsta este private in baza
    a.nume = "Animal X"; // eroare: nume este protected in baza
    a.greutate = 3.2; // OK: greutate este public in baza

    Caine c;
    c.varsta = 3; // eroare: varsta nu este accesibil in derivata
    c.nume = "Caine Y"; // eroare: nume este private in derivata
    c.greutate = 4.7; // OK: greutate este public in derivata
    c.talie = "mare"; // eroare: talie este private in derivata
    c.blana_lunga = true; // eroare: blana_lunga este protected in
    derivata
    c.lungime_coadă = 23; // OK: lungime_coadă este public in derivata

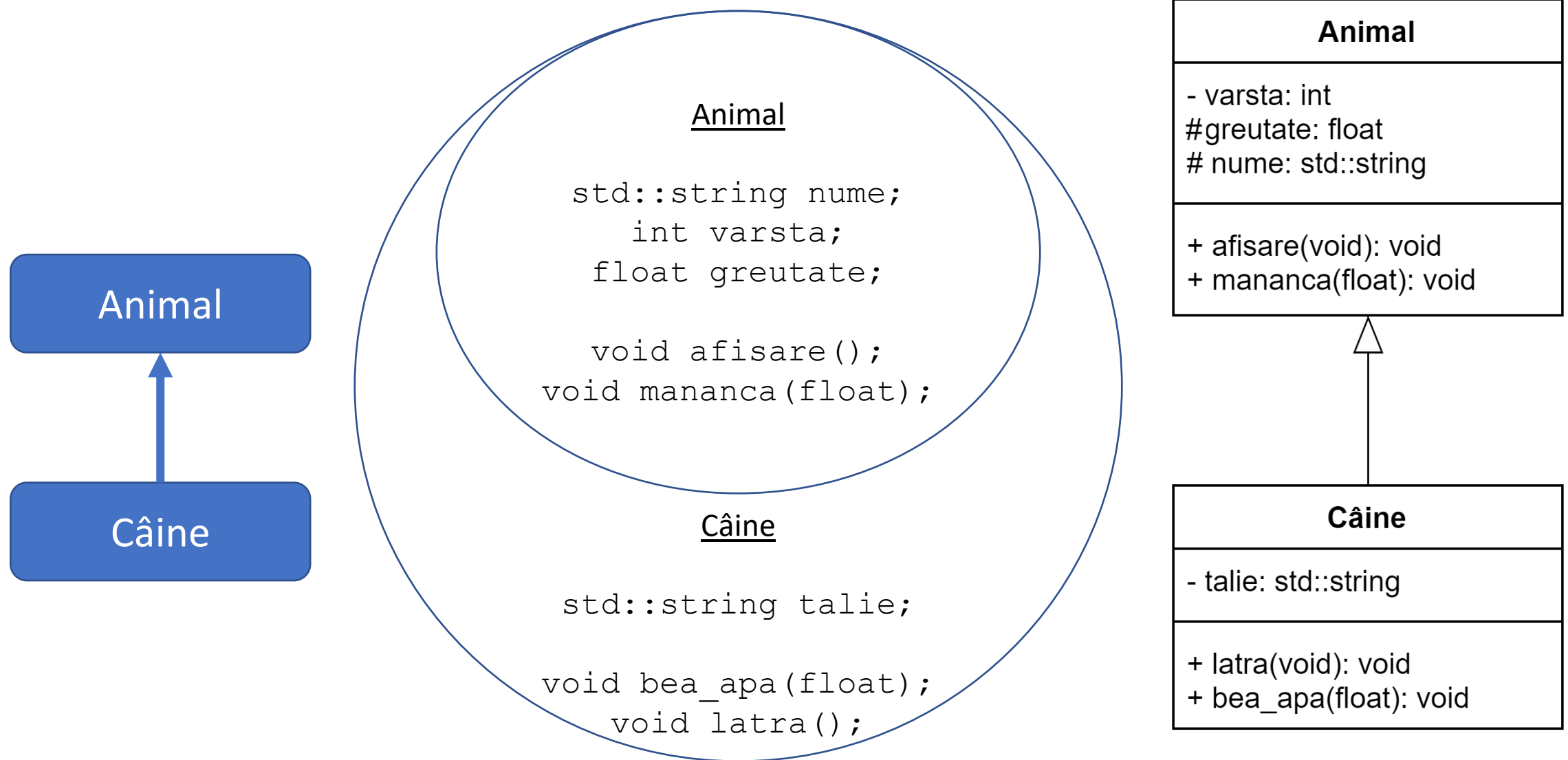
    return 0;
}
```

Concepte de bază

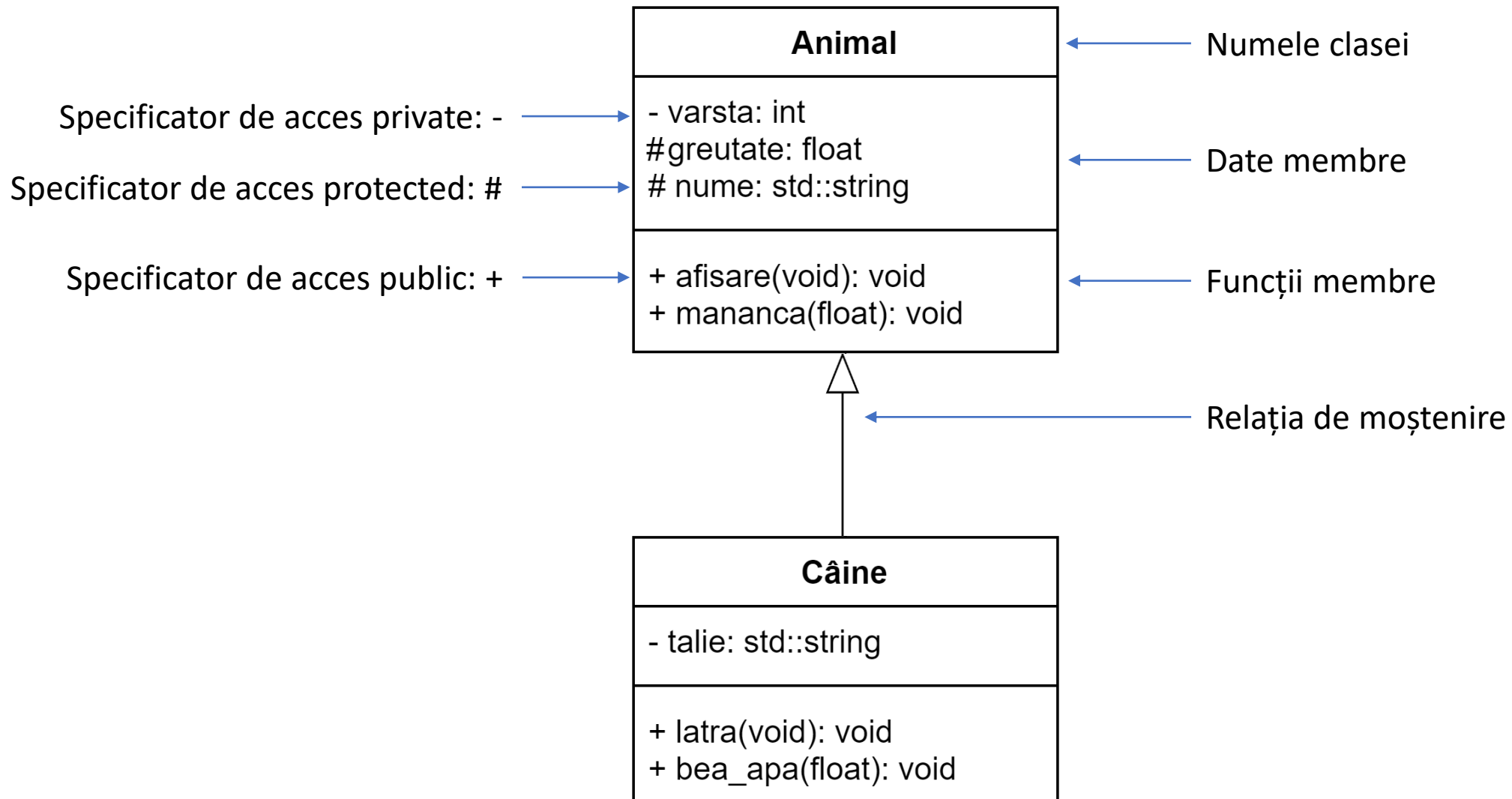
O clasă derivată moștenește de la clasa de bază toți membrii mai puțin:

- Constructorii și destructorii;
- Operatorul de asignare (`operator=`);
- “Prietenii” săi;
- Membrii privați.

Reprezentare grafică



UML (Unified Modeling Language)



Ordinea apelării constructorilor/destructorilor

Ordinea apelării constructorilor

O clasă derivată preia de la clasa de bază toți membrii (cu excepțiile menționate) => pentru crearea unei instanțe din derivată trebuie să fie instanțiată, mai întâi, clasa de bază prin apelarea constructorului său.

Într-un lanț de moșteniri succesive, ordinea apelului constructorilor este de la clasa de bază spre clasa cea mai derivată.

Ordinea apelării destructorilor

O clasă derivată conține toți membrii bazei sale, la care mai adaugă o serie de membri particulari, necunoscuți clasei de bază. Fiecare destructor se va ocupa de a șterge datele membre ale clasei din care face parte.

Într-un lanț de moșteniri succesive, ordinea apelului destructorilor este de la clasa cea mai derivată spre clasa de bază.

Ordinea apelării constructorilor/destructorilor

```
#include <iostream>

class Baza{
public:
    Baza(){
        std::cout << "Constructor Baza" << std::endl;
    }
    ~Baza(){
        std::cout << "Destructor Baza" << std::endl;
    }
};

class Derivata: public Baza{
public:
    Derivata(){
        std::cout << "Constructor Derivata" << std::endl;
    }
    ~Derivata(){
        std::cout << "Destructor Derivata" << std::endl;
    }
};

int main(){
    Derivata d;
    return 0;
}
```

```
Constructor Baza
Constructor Derivata
Destructor Derivata
Destructor Baza

Process returned 0 (0x0)   execution time : 0.285 s
Press any key to continue.
```

Ascunderea membrilor

Ascunderea membrilor

Definirea în clasa derivată a unui membru (dată/funcție) cu același nume cu un membru din clasa de bază duce la ascunderea membrului din clasa de bază.

Accesarea membrilor ascunși din clasa de bază se poate face prin calificarea completă a numelui membrului în momentul utilizării.

Aducerea membrului ascuns din clasa de bază în același domeniu de definiție (scope) cu clasa derivată se face folosind sintaxa `using`.

Ascunderea datelor membre

```
#include <iostream>
```

```
class Animal{  
public:  
    float greutate;  
    void afisare(){  
        std::cout << "Animal::afisare()" << std::endl;  
        std::cout << greutate << std::endl;  
    }  
};
```

```
class Caine : public Animal{  
public:  
    float greutate;
```

← float greutate ascunde data membră Animal::greutate

```
int main(){  
    Caine c;  
    c.greutate = 5;  
    c.afisare();  
  
    return 0;  
}
```

← Inițializare Caine::greutate cu valoarea 5

← Afișare Animal::greutate – valoare nedefinită

Caine::greutate și Animal::greutate sunt două variabile diferite, cu același nume, însă domenii de definiție diferite

Ascunderea datelor membre

```
#include <iostream>

class Animal{
public:
    float greutate;
    void afisare(){
        std::cout << "Animal::afisare()" << std::endl;
        std::cout << greutate << std::endl;
    }
};

class Caine : public Animal{
public:
    float greutate; ← float greutate ascunde data membră Animal::greutate
};

int main(){
    Caine c;
    c.Animal::greutate = 5; ← Inițializare Animal::greutate cu valoarea 5
    c.afisare(); ← Afișare Animal::greutate – valoarea 5
    return 0;
}
```

Ascunderea datelor membre

```
#include <iostream>
```

```
class Animal{  
protected:  
    float greutate;  
public:  
    void afisare(){  
        std::cout << "Animal::afisare()" << std::endl;  
        std::cout << greutate << std::endl;  
    }  
};
```

```
class Caine : public Animal{  
public:  
    using Animal::greutate;  
};
```

Aducerea datei membre `Animal::greutate` în domeniul de definiție al clasei `Caine` și schimbarea specificatorului de acces din `protected` în `public`

```
int main(){  
    Caine c;  
    c.greutate = 5;  
    c.afisare();  
  
    return 0;  
}
```

Inițializare `Animal::greutate` cu valoarea 5

Afișare `Animal::greutate` – valoarea 5

Ascunderea funcțiilor membre

```
#include <iostream>

class Animal{
private:
    int varsta;
protected:
    float greutate;
    std::string nume;
public:
    void afisare(){
        std::cout << "Animal::afisare()" << std::endl;
        std::cout << varsta << std::endl;
        std::cout << nume << std::endl;
        std::cout << greutate << std::endl;
    }
};
```

```
class Caine : public Animal{
private:
    std::string talie;
public:
    void afisare(){
        std::cout << "Caine::afisare()" << std::endl;
        std::cout << talie << std::endl << std::endl;
    }
};
```

```
int main(){
    Caine c;
    c.afisare(); // apeleaza Caine::afisare()
    c.Animal::afisare(); // apeleaza Animal::afisare()

    return 0;
}
```

Caine::afisare() ascunde funcția Animal::afisare()

Ascunderea funcțiilor membre

```
#include <iostream>

class Animal{
private:
    int varsta;
protected:
    float greutate;
    std::string nume;
public:
    void afisare(){
        std::cout << "Animal::afisare()" << std::endl;
        std::cout << varsta << std::endl;
        std::cout << nume << std::endl;
        std::cout << greutate << std::endl;
    }
};
```

```
class Caine : public Animal{
private:
    std::string talie;
public:
    void afisare(int a){
        std::cout << "Caine::afisare()" << std::endl;
        std::cout << talie << std::endl << std::endl;
    }
};
```

```
int main(){
    Caine c;
    c.afisare(); // eroare: no matching function for
                // call to 'Caine::afisare()'
    c.afisare(1); // apeleaza Caine::afisare(int)

    return 0;
}
```

Caine::afisare(int) ascunde funcția Animal::afisare()

Ascunderea funcțiilor membre

```
#include <iostream>

class Animal{
private:
    int varsta;
protected:
    float greutate;
    std::string nume;
public:
    void afisare(){
        std::cout << "Animal::afisare()" << std::endl;
        std::cout << varsta << std::endl;
        std::cout << nume << std::endl;
        std::cout << greutate << std::endl;
    }
};

class Caine : public Animal{
private:
    std::string talie;
public:
    using Animal::afisare; ← Se aduce în domeniul de definiție funcția Animal::afisare()
    void afisare(int a){
        std::cout << "Caine::afisare()" << std::endl;
        std::cout << talie << std::endl << std::endl;
    }
};
```

```
int main(){
    Caine c;
    c.afisare(); // apeleaza Animal::afisare()
    c.afisare(1); // apeleaza Caine::afisare(int)

    return 0;
}
```

Suprascrierea funcțiilor (function overriding)

Suprascrierea funcțiilor

```
#include <iostream>
class Animal {
public:
    void spune_ceva() {std::cout << "Animal vorbitor" << std::endl;}
};
class Caine : public Animal {
public:
    void spune_ceva() {std::cout << "Caine vorbitor" << std::endl;}
};
int main() {
    Animal* pa1 = new Animal();
    Caine* pc = new Caine();

    pa1->spune_ceva();
    pc->spune_ceva();

    return 0;
}
```

```
Animal vorbitor
Caine vorbitor

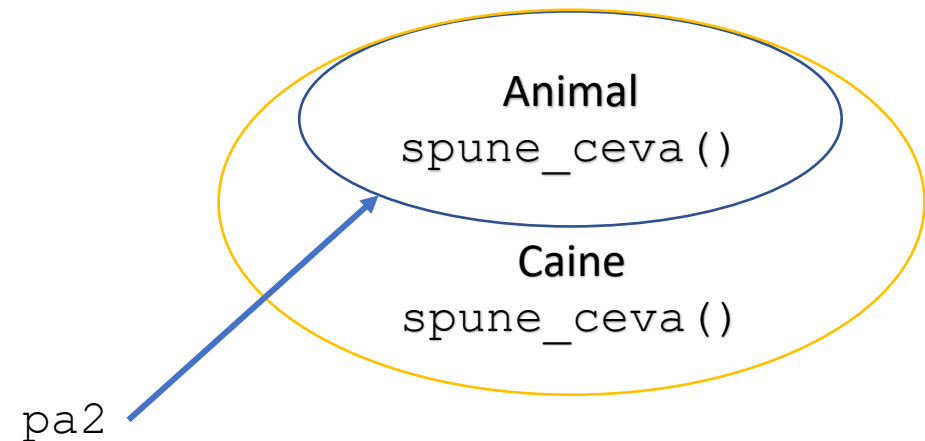
Process returned 0 (0x0)   execution time : 0.042 s
Press any key to continue.
```

Suprascrierea funcțiilor

```
#include <iostream>
class Animal {
public:
    void spune_ceva() {std::cout << "Animal vorbitor" << std::endl;}
};
class Caine : public Animal {
public:
    void spune_ceva() {std::cout << "Caine vorbitor" << std::endl;}
};
int main() {
    Animal* pa1 = new Animal();
    Caine* pc = new Caine();
    → Animal* pa2 = new Caine();

    pa1->spune_ceva();
    pc->spune_ceva();
    → pa2->spune_ceva();

    return 0;
}
```



Suprascrierea funcțiilor

```
#include <iostream>
class Animal {
public:
    void spune_ceva() {std::cout << "Animal vorbitor" << std::endl;}

};
class Caine : public Animal {
public:
    void spune_ceva() {std::cout << "Caine vorbitor" << std::endl;}

};
int main() {
    Animal* pa1 = new Animal();
    Caine* pc = new Caine();
    → Animal* pa2 = new Caine();


    pa1->spune_ceva();
    pc->spune_ceva();
    → pa2->spune_ceva();

    return 0;
}
```

```
Animal vorbitor
Caine vorbitor
Animal vorbitor

Process returned 0 (0x0)   execution time : 0.028 s
Press any key to continue.
```

Suprascrierea funcțiilor

 Dacă o **funcție (ne-virtuală)** din clasa de bază este redefinită în clasa derivată și apelul ei se va face prin intermediul unui pointer/referință la un (sub)obiect, atunci se va apela funcția din clasa corespondentă tipului **pointer-ului/referinței**.

La momentul compilării se cunoaște asocierea dintre obiecte și funcțiile apelate => se realizează legarea statică (early/static binding).

Suprascrierea funcțiilor

```
#include <iostream>
class Animal {
public:
    void spune_ceva() {std::cout << "Animal vorbitor" << std::endl;}
    → virtual void spune_ceva2() {std::cout << "[virtual] Animal vorbitor" << std::endl;}
};
class Caine : public Animal {
public:
    void spune_ceva() {std::cout << "Caine vorbitor" << std::endl;}
    → void spune_ceva2() {std::cout << "[virtual] Caine vorbitor" << std::endl;}
};
int main() {
    Animal* pa1 = new Animal();
    Caine* pc = new Caine();
    Animal* pa2 = new Caine();

    pa1->spune_ceva();
    pc->spune_ceva();
    pa2->spune_ceva();

    → { pa1->spune_ceva2();
        pc->spune_ceva2();
        pa2->spune_ceva2();
    }

    return 0;
}
```


Suprascrierea funcțiilor

```
#include <iostream>
class Animal {
public:
    void spune_ceva() {std::cout << "Animal vorbitor" << std::endl;}
    → virtual void spune_ceva2() {std::cout << "[virtual] Animal vorbitor" << std::endl;}
};
class Caine : public Animal {
public:
    void spune_ceva() {std::cout << "Caine vorbitor" << std::endl;}
    → void spune_ceva2() {std::cout << "[virtual] Caine vorbitor" << std::endl;}
};
int main() {
    Animal* pa1 = new Animal();
    Caine* pc = new Caine();
    Animal* pa2 = new Caine();

    pa1->spune_ceva();
    pc->spune_ceva();
    pa2->spune_ceva();

    → { pa1->spune_ceva2();
        pc->spune_ceva2();
        pa2->spune_ceva2();
    }


    return 0;
}
```

03/04/22

```
Animal vorbitor
Caine vorbitor
Animal vorbitor
[virtual] Animal vorbitor
[virtual] Caine vorbitor
[virtual] Caine vorbitor

Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

Suprascrierea funcțiilor

 Dacă o **funcție virtuală** din clasa de bază este redefinită în clasa derivată și apelul ei se va face prin intermediul unui pointer/referință la un (sub)obiect, atunci se va apela funcția din clasa corespondentă tipului **(sub)obiectului** către care indică pointerul/referința.

La momentul compilării nu se cunoaște asocierea dintre obiecte și funcțiile apelate. Aceasta devine cunoscută în momentul rulării programului => se realizează legarea dinamică (late/dynamic binding).

Suprascrierea funcțiilor

```
#include <iostream>
class Animal {
public:
    void spune_ceva() { std::cout << "Animal vorbitor" << std::endl; }
    virtual void spune_ceva2() { std::cout << "[virtual] Animal vorbitor" << std::endl; }
};
class Caine : public Animal {
public:
    void spune_ceva() { std::cout << "Caine vorbitor" << std::endl; }
    void spune_ceva2() { std::cout << "[virtual] Caine vorbitor" << std::endl; }
};
```

```
void fun_1(Animal *ptr) { ptr->spune_ceva(); }
void fun_2(Animal *ptr) { ptr->spune_ceva2(); }
```

Funcțiile primesc ca argument
pointer către clasa de bază

```
int main() {
    Animal* pa = new Animal();
    Caine* pc = new Caine();

    fun_1(pa);
    fun_1(pc);

    fun_2(pa);
    fun_2(pc);
    return 0;
}
```

```
Animal vorbitor
Animal vorbitor
[virtual] Animal vorbitor
[virtual] Caine vorbitor
```

```
Process returned 0 (0x0)   execution time : 0.009 s
Press any key to continue.
```

Suprascrierea funcțiilor

```
#include <iostream>
class Animal {
public:
    virtual void talk(){std::cout << "Animal vorbitor" << std::endl;}
};
class Caine : public Animal {
public:
    void talk(){std::cout << "Caine vorbitor" << std::endl;}
};
class Caine_maidanez : public Caine {
public:
    void talk(){std::cout << "Caine maidanez vorbitor" << std::endl;}
};

void fun(Animal *ptr) {ptr->talk();}

int main() {
    Animal* pa1 = new Animal();
    Animal* pa2 = new Caine();
    Animal* pa3 = new Caine_maidanez();

    fun(pa1);
    fun(pa2);
    fun(pa3);

    return 0;
}
```

Cuvântul cheie `virtual` este
moștenit implicit în clasele derivate

```
Animal vorbitor
Caine vorbitor
Caine maidanez vorbitor

Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

Suprascrierea funcțiilor

```
#include <iostream>
class Animal {
public:
    virtual void talk() {std::cout << "Animal vorbitor" << std::endl;}
};
class Caine : public Animal {
public:
    virtual void talk() {std::cout << "Caine vorbitor" << std::endl;}
};
class Caine_maidanez : public Caine {
public:
    virtual void talk() {std::cout << "Caine maidanez vorbitor" << std::endl;}
};

void fun(Animal *ptr) {ptr->talk();}

int main() {
    Animal* pa1 = new Animal();
    Animal* pa2 = new Caine();
    Animal* pa3 = new Caine_maidanez();

    fun(pa1);
    fun(pa2);
    fun(pa3);

    return 0;
}
```

Nu este nicio problemă dacă
adăugăm explicit cuvântul
virtual la funcțiile suprascrise.

```
Animal vorbitor
Caine vorbitor
Caine maidanez vorbitor

Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

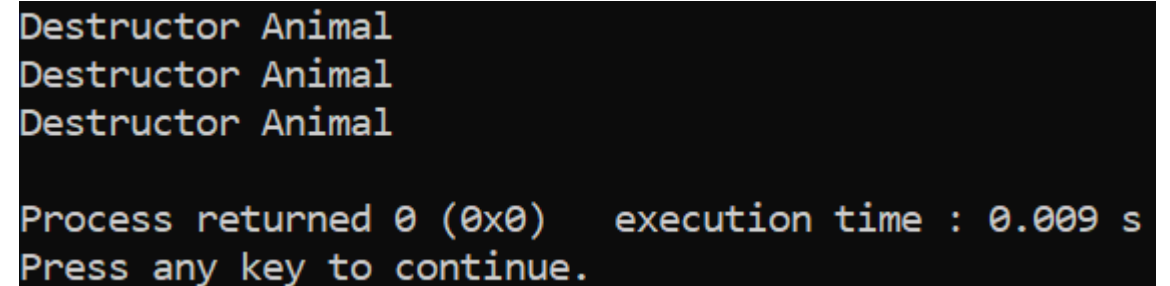
Suprascrierea funcțiilor – destructori virtuali

```
#include <iostream>
class Animal {
public:
    ~Animal() { std::cout << "Destructor Animal" << std::endl; }
};
class Caine : public Animal {
public:
    ~Caine() { std::cout << "Destructor Caine" << std::endl; }
};
class Caine_maidanez : public Caine {
public:
    ~Caine_maidanez() { std::cout << "Destructor Caine_maidanez" << std::endl; }
};

int main() {
    Animal* pa = new Animal();
    Animal* pc = new Caine();
    Animal* pcm = new Caine_maidanez();

    delete pa;
    delete pc;
    delete pcm;

    return 0;
}
```

A screenshot of a terminal window with a black background and white text. It shows the output of a C++ program. The first three lines are "Destructor Animal", "Destructor Animal", and "Destructor Animal", each on a new line. The fourth line is "Process returned 0 (0x0) execution time : 0.009 s". The fifth line is "Press any key to continue.".

```
Destructor Animal
Destructor Animal
Destructor Animal

Process returned 0 (0x0) execution time : 0.009 s
Press any key to continue.
```

Lipsa destructorilor virtuali poate cauza comportament nedefinit

Suprascrierea funcțiilor – destructori virtuali

```
#include <iostream>
class Animal {
public:
    virtual ~Animal() { std::cout << "Destructor Animal" << std::endl; }
};
class Caine : public Animal {
public:
    ~Caine() { std::cout << "Destructor Caine" << std::endl; }
};
class Caine_maidanez : public Caine {
public:
    ~Caine_maidanez() { std::cout << "Destructor Caine_maidanez" << std::endl; }
};

int main() {
    Animal* pa = new Animal();
    Animal* pc = new Caine();
    Animal* pcm = new Caine_maidanez();

    delete pa;
    delete pc;
    delete pcm;

    return 0;
}
```

```
Destructor Animal
Destructor Caine
Destructor Animal
Destructor Caine_maidanez
Destructor Caine
Destructor Animal

Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

Suprascrierea funcțiilor – specificatorul `override`

Specifică faptul că o funcție virtuală suprascrive o altă funcție virtuală.

- Apare după declararea funcției.
- Asigură că funcția este virtuală și că suprascrive o funcție virtuală din clasa de bază. Generează o eroare de sintaxă în caz contrar.
- Se poate aplica și destructorilor.
- Ajută la o mai bună înțelegere a codului.

Suprascrierea funcțiilor – specificatorul `override`

```
#include <iostream>

class Baza {
public:
    void f(){ std::cout << "Baza::f()" << std::endl;}
    void f2(){ std::cout << "Baza::f2()" << std::endl;}
};

class Derivata: public Baza {
public:
    void f() override {std::cout << "Derivata::f()" << std::endl;}
    void f2() override;
};

void Derivata::f2(){std::cout << "Derivata::f2()" << std::endl;}

int main(){
    Baza *d = new Derivata();
    d->f();
    d->f2();
    return 0;
}
```

11 error: 'void Derivata::f()' marked 'override', but does not override
12 error: 'void Derivata::f2()' marked 'override', but does not override
=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

Suprascrierea funcțiilor – specificatorul `override`

```
#include <iostream>

class Baza {
public:
    virtual void f(){ std::cout << "Baza::f()" << std::endl;}
    virtual void f2(){ std::cout << "Baza::f2()" << std::endl;}
};

class Derivata: public Baza {
public:
    void f() override {std::cout << "Derivata::f()" << std::endl;}
    void f2() override;
};

void Derivata::f2(){std::cout << "Derivata::f2()" << std::endl;}

int main(){
    Baza *d = new Derivata();
    d->f();
    d->f2();
    return 0;
}
```

```
Derivata::f()
Derivata::f2()

Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```


Suprascrierea funcțiilor – specificatorul `override`

```
#include <iostream>

class Baza {
public:
    ~Baza() {std::cout << "Destructor Baza" << std::endl;}
};

class Derivata: public Baza {
public:
    ~Derivata() override {std::cout << "Destructor Derivata" << std::endl;}
};

int main() {
    Baza *d = new Derivata();
    delete d;
    return 0;
}
```



```
10 error: 'Derivata::~~Derivata()' marked 'override', but does not override
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Suprascrierea funcțiilor – specificatorul `override`

```
#include <iostream>

class Baza {
public:
    virtual ~Baza() {std::cout << "Destructor Baza" << std::endl;}
};


class Derivata: public Baza {
public:
    ~Derivata() override {std::cout << "Destructor Derivata" << std::endl;}
};

int main(){
    Baza *d = new Derivata();
    delete d;
    return 0;
}
```

```
Destructor Derivata
Destructor Baza

Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```

Suprascrierea funcțiilor – specificatorul `final`

 **def** După declaratorul unei funcții virtuale – specifică faptul că o funcție virtuală nu poate fi suprascrisă într-o clasă derivată din clasa din care respectiva funcție face parte.

 **def** După declaratorul unei clase – specifică faptul că o clasă nu mai poate fi derivată.

Suprascrierea funcțiilor – specificatorul `final`

```
#include <iostream>

class Baza {
public:
    virtual void foo() final {std::cout << "Baza::foo()" << std::endl;}
};

class Derivata: public Baza {
public:
    void foo() {std::cout << "Derivata::foo()" << std::endl;}
};

int main(){
    Baza *d = new Derivata();
    d->foo();
    return 0;
}
```

```
10 error: virtual function 'virtual void Derivata::foo()' overriding final function
5 note: overridden function is 'virtual void Baza::foo()'
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Suprascrierea funcțiilor – specificatorul `final`

```
#include <iostream>

class Baza final {
public:
    void foo() {std::cout << "Baza::foo()" << std::endl;}
};

class Derivata: public Baza {
public:
    void foo() {std::cout << "Derivata::foo()" << std::endl;}
};

int main(){
    Baza *d = new Derivata();
    d->foo();
    return 0;
}
```

```
8 error: cannot derive from 'final' base 'Baza' in derived type 'Derivata'
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Suprascrierea funcțiilor – funcții pur virtuale



Funcțiile *pur virtuale* sunt funcții virtuale care **trebuie** suprascrise în clasele derivate.

- O funcție virtuală devine **pur virtuală** dacă i se atribuie valoarea 0 după declarație.
- O funcție pur virtuală nu poate fi definită în continuare declarației (în interiorul clasei). Definiția ei (dacă există) trebuie să aibă loc în afara clasei.
- O clasă care conține cel puțin o funcție pur virtuală devine **abstractă**. Clasele abstracte nu pot fi instanțiate! Se mai numesc și **interfețe**.
- Dacă o clasă moștenește o clasă abstractă și nu suprascrie toate funcțiile pur virtuale, atunci devine și ea abstractă.

Suprascrierea funcțiilor – funcții pur virtuale

```
#include <iostream>

class Baza { // clasa abstracta
public:
    virtual void foo() = 0; // functie pur virtuala
};

class Derivata: public Baza {
public:
    void foo() {std::cout << "Derivata::foo()" << std::endl;}
};

int main(){
    Baza *d = new Derivata();
    d->foo();
    return 0;
}
```

```
Derivata::foo()
```

```
Process returned 0 (0x0)   execution time : 0.006 s
Press any key to continue.
```

Suprascrierea funcțiilor – funcții pur virtuale

```
#include <iostream>

class Baza { // clasa abstracta
public:
    virtual void foo() = 0; // functie pur virtuala
};

void Baza::foo() {std::cout << "Baza::foo()" << std::endl;}

class Derivata: public Baza {
public:
    void foo() {
        Baza::foo();
        std::cout << "Derivata::foo()" << std::endl;
    }
};

int main() {
    Baza *d = new Derivata();
    d->foo();
    return 0;
}
```

```
Baza::foo()
Derivata::foo()

Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

Suprascrierea funcțiilor – funcții pur virtuale

```
#include <iostream>

class Baza { // clasa abstracta
public:
    virtual void foo() = 0; // functie pur virtuala
};

void Baza::foo() {std::cout << "Baza::foo()" << std::endl;}

class Derivata: public Baza {
public:
    void foo() {
        Baza::foo();
        std::cout << "Derivata::foo()" << std::endl;
    }
};

int main(){
    Baza *d = new Baza();
    d->foo();
    return 0;
}
```

```
19 error: invalid new-expression of abstract class type 'Baza'
3  note:   because the following virtual functions are pure within 'Baza':
8  note:       'virtual void Baza::foo()'
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Suprascrierea funcțiilor – funcții pur virtuale

```
#include <iostream>

class Baza { // clasa abstracta
public:
    virtual void foo() = 0; // functie pur virtuala
};

class Derivata: public Baza {}; // nu suprascrie functia pur virtuala => clasa abstracta

class Derivata_din_nou: public Derivata {
public:
    void foo() {std::cout << "Derivata_din_nou::foo()" << std::endl;}
};

int main(){
    Baza *d = new Derivata();
    d->foo();
    return 0;
}
```

```
18 error: invalid new-expression of abstract class type 'Derivata'
10 note: because the following virtual functions are pure within 'Derivata':
8 note: 'virtual void Baza::foo()'
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Suprascrierea funcțiilor – funcții pur virtuale

```
#include <iostream>

class Baza { // clasa abstracta
public:
    virtual void foo() = 0; // functie pur virtuala
};

class Derivata: public Baza {}; // nu suprascrie functia pur virtuala => clasa abstracta

class Derivata_din_nou: public Derivata {
public:
    void foo() {std::cout << "Derivata_din_nou::foo()" << std::endl;}
};

int main(){
    Baza *d = new Derivata_din_nou();
    d->foo();
    return 0;
}
```

```
Derivata_din_nou::foo()

Process returned 0 (0x0)   execution time : 0.027 s
Press any key to continue.
```

Tipuri diferite de moșteniri

Tipuri de moșteniri (inheritance):

1. Unică (single)
2. Multiplă (multiple)
3. Multinivel (multilevel)
4. Ierarhică (hierarchical)
5. Hibridă (hybrid)
6. Multicale (multipath)

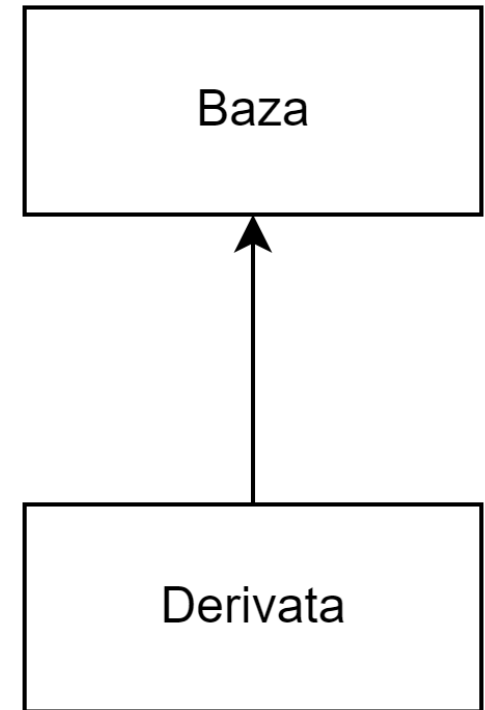
1. Moștenire unică (single inheritance)

```
#include <iostream>

class Baza{
public:
    int a;
};

class Derivata: public Baza{
public:
    int b;
};

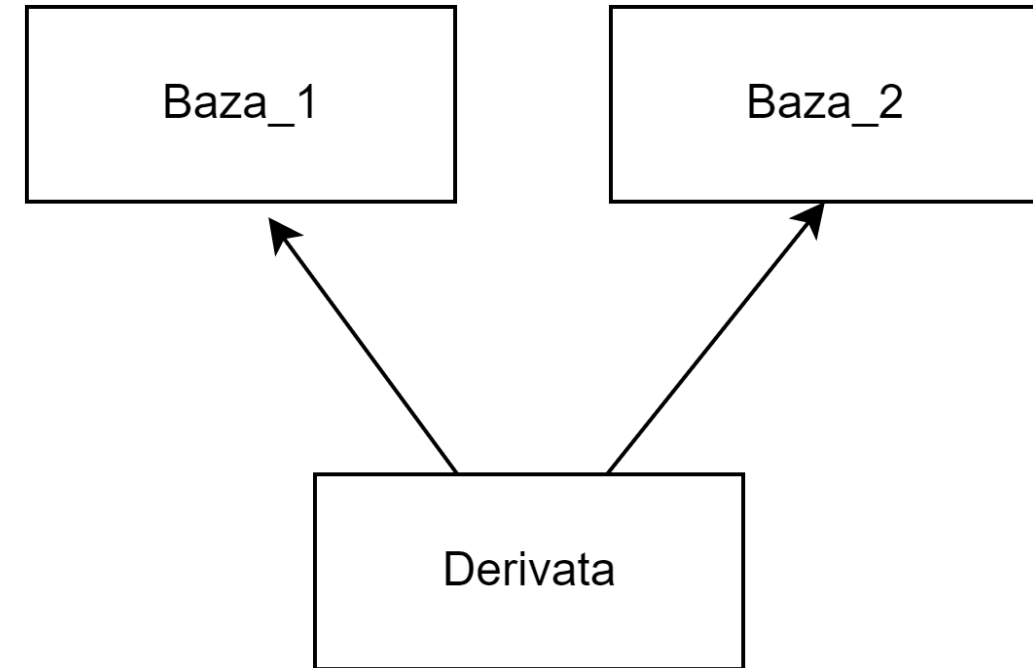
int main(){
    Derivata d;
    d.a = 5;
    d.b = 10;
    std::cout << d.a << " " << d.b << std::endl;
    return 0;
}
```



2. Moștenire multiplă (multiple inheritance)

```
#include <iostream>
class Baza_1{
public:
    Baza_1(){
        std::cout << "Constructor Baza_1" << std::endl;
    }
};
class Baza_2{
public:
    Baza_2(){
        std::cout << "Constructor Baza_2" << std::endl;
    }
};
class Derivata: 1 public Baza_1, 2 public Baza_2{
public:
3    Derivata(){
        std::cout << "Constructor Derivata" << std::endl;
    }
};

int main(){
    Derivata d;
    return 0;
}
```



```
Constructor Baza_1
Constructor Baza_2
Constructor Derivata
```

```
Process returned 0 (0x0)   execution time : 0.065 s
Press any key to continue.
```

2. Moștenire multiplă (multiple inheritance)

```
#include <iostream>
class Baza_1{
public:
    void display(){
        std::cout << "Baza_1::display()" << std::endl;
    }
};
class Baza_2{
public:
    void display(){
        std::cout << "Baza_2::display()" << std::endl;
    }
};
class Derivata: public Baza_1, public Baza_2{};

int main(){
    Derivata d;
    d.display();
    return 0;
}
```

Problemă de ambiguitate



```
23 error: request for member 'display' is ambiguous
10 note: candidates are: 'void Baza_2::display()'
4  note:                  'void Baza_1::display()'
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

2. Moștenire multiplă (multiple inheritance)

```
#include <iostream>
class Baza_1{
public:
    void display(){
        std::cout << "Baza_1::display()" << std::endl;
    }
};
class Baza_2{
public:
    void display(){
        std::cout << "Baza_2::display()" << std::endl;
    }
};
class Derivata: public Baza_1, public Baza_2{

};

int main(){
    Derivata d;
    d.Baza_1::display();
    d.Baza_2::display();
    return 0;
}
```

Aceeași soluție ca în cazul ascunderii numelor membrilor (slide-urile 19-26)

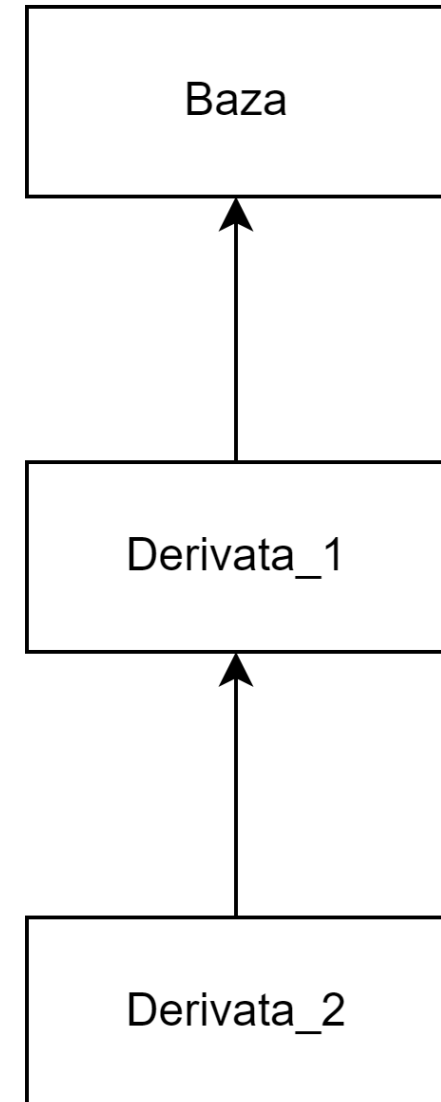
```
Baza_1::display()
Baza_2::display()
```

```
Process returned 0 (0x0)   execution time : 0.009 s
Press any key to continue.
```

3. Moștenire multinivel (multilevel inheritance)

```
#include <iostream>
class Baza{
public:
    int a;
};
class Derivata_1: public Baza{
public:
    int b;
};
class Derivata_2: public Derivata_1{
public:
    int c;
};

int main(){
    Derivata_2 d;
    d.a = 5;
    d.b = 10;
    d.c = 17;
    return 0;
}
```



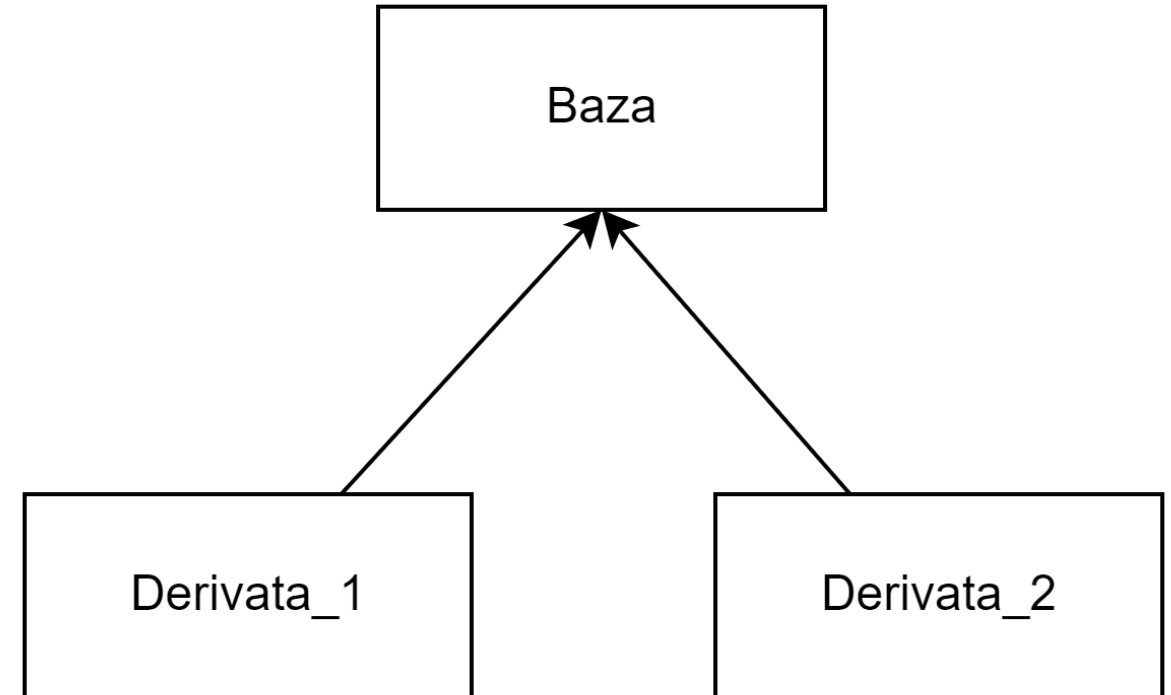
4. Moștenire ierarhică (hierarchical inheritance)

```
#include <iostream>
class Baza{
public:
    int a;
};
class Derivata_1: public Baza{
public:
    int b;
};
class Derivata_2: public Baza{
public:
    int c;
};

int main(){
    Derivata_1 d1;
    d1.a = 5;
    d1.b = 10;

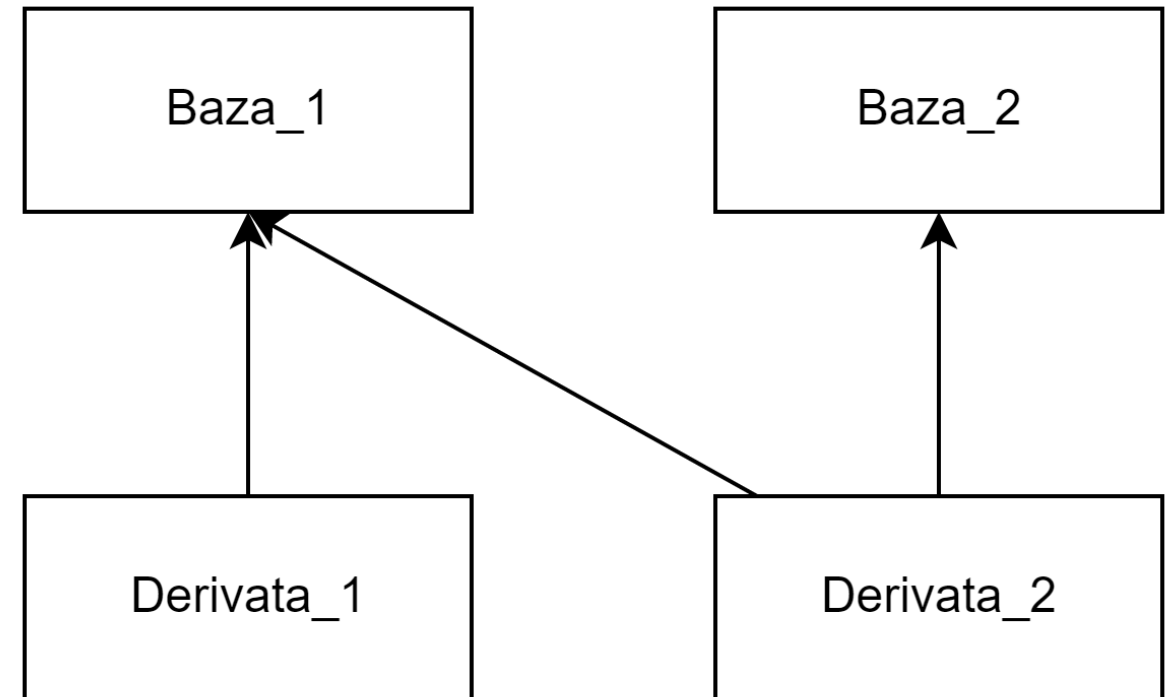
    Derivata_2 d2;
    d2.a = 1;
    d2.c = 17;

    return 0;
}
```



5. Moștenire hibridă (hybrid inheritance)

```
#include <iostream>
class Baza_1{
public:
    int a;
};
class Baza_2{
public:
    int b;
};
class Derivata_1: public Baza_1{
public:
    int c;
};
class Derivata_2: public Baza_1, public Baza_2{
public:
    int d;
};
int main(){
    Derivata_1 d1;
    d1.a = 5;
    d1.c = 10;
    Derivata_2 d2;
    d2.a = 1;
    d2.b = 17;
    d2.d = 20;
    return 0;
}
```



6. Moștenire multicală (multipath inheritance)

```
#include <iostream>
class Baza{
public:
    int a;
};
class Derivata_1:public Baza{};
class Derivata_2: public Baza{};
class Derivata_3: public Derivata_1, public Derivata_2{};

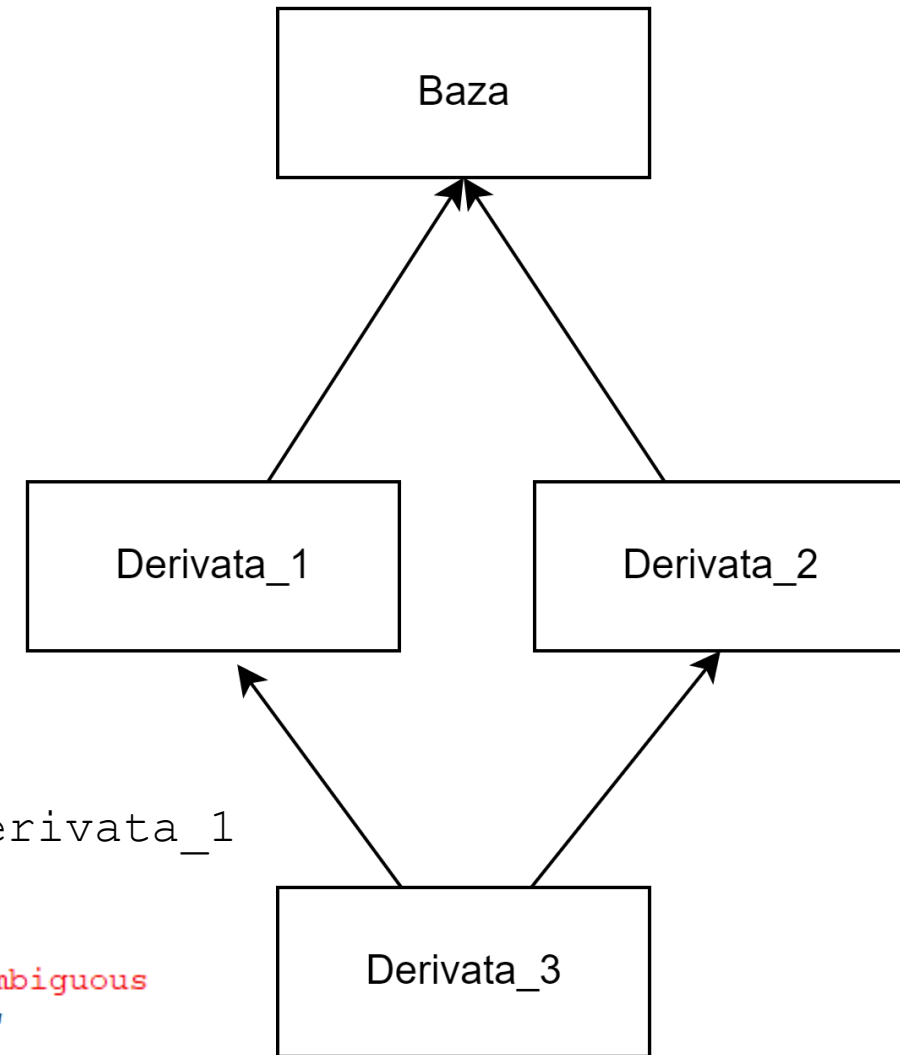
int main(){
    Derivata_1 d1;
    d1.a = 5;

    Derivata_2 d2;
    d2.a = 10;

    Derivata_3 d3;
    d3.a = 0;
    return 0;
}
```

Este vorba despre variabila a moștenită prin clasa Derivata_1 sau variabila a moștenită prin clasa Derivata_2?

```
16 error: request for member 'a' is ambiguous
4 note: candidates are: 'int Baza::a'
4 note:                  'int Baza::a'
```



6. Moștenire multicale (multipath inheritance)

```
#include <iostream>
class Baza{
public:
    int a;
};
class Derivata_1:public Baza{};
class Derivata_2: public Baza{};
class Derivata_3: public Derivata_1, public Derivata_2{};

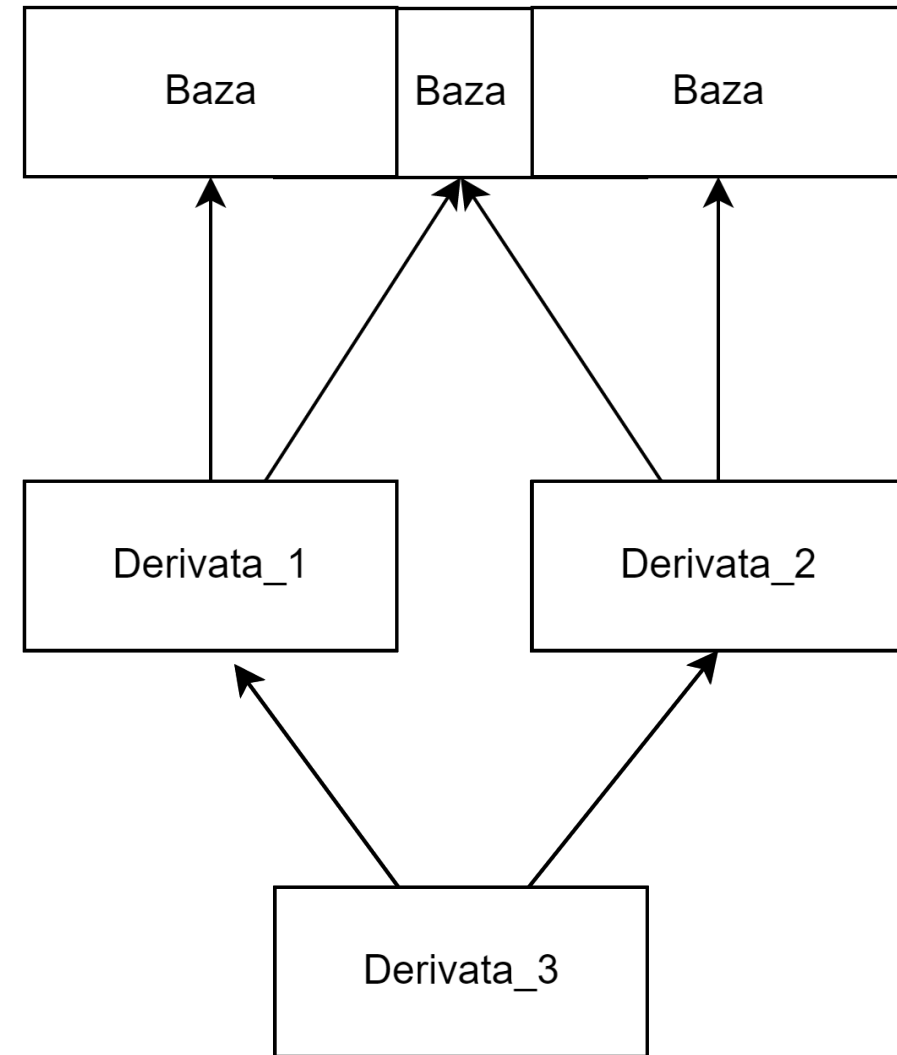
int main(){
    Derivata_1 d1;
    d1.a = 5;

    Derivata_2 d2;
    d2.a = 10;

    Derivata_3 d3;
    d3.Derivata_1::a = 0;
    d3.Derivata_2::a = 3;

    return 0;
}
```

Compilatorul vede 2 copii diferite ale
variabilei a: d3.Derivata_1::a,
respectiv d3.Derivata_2::a



6. Moștenire multicală (multipath inheritance)

```
#include <iostream>
class Baza{
public:
    int a;
};
class Derivata_1:public Baza{};
class Derivata_2: public Baza{};
class Derivata_3: public Derivata_1, public Derivata_2{};

int main(){
    Derivata_1 d1;
    d1.a = 5;

    Derivata_2 d2;
    d2.a = 10;

    Derivata_3 d3;
    d3.Derivata_1::a = 0;
    d3.Derivata_2::a = 3;

    return 0;
}
```

Compilatorul vede 2 copii diferite ale
variabilei a: d3.Derivata_1::a,
respectiv d3.Derivata_2::a

6. Moștenire multicală (multipath inheritance)

```
#include <iostream>
class B{
public:
    int a=123;
    B(){std::cout<<"B::B() "<<std::endl;}
    B(int a):a(a){std::cout<<"B::B ("<<a<<" "<<std::endl;}
    int get_a(){return a;}
};
class D1: public B{
public:
    D1(int x):B(x){std::cout<<"D1::D1 ("<<x<<" "<<std::endl;}
};
class D2: public B{
public:
    D2(int x):B(x){std::cout<<"D2::D2 ("<<x<<" "<<std::endl;}
};
class D3: public D1, public D2{
public:
    D3(int x,int y):D1(x),D2(y){std::cout<<"D3::D3 ("<<x<<","<<y<<" "<<std::endl;}
};

int main(){
    D3 d(5,6);
    //    std::cout<<d.get_a()<<std::endl; // ambiguous request
    std::cout<<d.D1::get_a()<<std::endl;
    std::cout<<d.D2::get_a()<<std::endl;
    return 0;
}
```

```
B::B(5)
D1::D1(5)
B::B(6)
D2::D2(6)
D3::D3(5, 6)
5
6
```

6. Moștenire multicală (multipath inheritance)

Soluția: moștenirea virtuală a clasei de bază comună. Astfel, clasa cea mai derivată (`Derivata_3`) va conține o singură copie a membrilor clasei de bază (`Baza`).

Clasa cea mai derivată (`Derivata_3`) este responsabilă de crearea subobiectului din clasa cea mai de bază (`Baza`), deoarece derivatele intermediare (`Derivata_1` și `Derivata_2`) nu vor face instanțierea acestuia.

6. Moștenire multicală (multipath inheritance)

```
#include <iostream>
class B{
public:
    int a=123;
    B(){std::cout<<"B::B() "<<std::endl;}
    B(int a):a(a){std::cout<<"B::B ("<<a<<" "<<std::endl;}
    int get_a(){return a;}
};
class D1: virtual public B{
public:
    D1(int x):B(x){std::cout<<"D1::D1 ("<<x<<" "<<std::endl;}
};
class D2: virtual public B{
public:
    D2(int x):B(x){std::cout<<"D2::D2 ("<<x<<" "<<std::endl;}
};
class D3: public D1, public D2{
public:
    D3(int x, int y):D1(x),D2(y){std::cout<<"D3::D3 ("<<x<<","<<y<<" "<<std::endl;}
};

int main(){
    D3 d(5,6);
    std::cout<<d.get_a()<<std::endl;
    return 0;
}
```

```
B::B()
D1::D1(5)
D2::D2(6)
D3::D3(5, 6)
123
```

Constructorii din D1 și D2 nu deleagă argumentul întreg către constructorul din B, deoarece moștenirea este virtuală.

6. Moștenire multicală (multipath inheritance)

```
#include <iostream>
class B{
public:
    int a=123;
    B(){std::cout<<"B::B() "<<std::endl;}
    B(int a):a(a){std::cout<<"B::B ("<<a<<" "<<std::endl;}
    int get_a(){return a;}
};
class D1: virtual public B{
public:
    D1(int x):B(x){std::cout<<"D1::D1 ("<<x<<" "<<std::endl;}
};
class D2: virtual public B{
public:
    D2(int x):B(x){std::cout<<"D2::D2 ("<<x<<" "<<std::endl;}
};
class D3: public D1, public D2{
public:
    D3(int x, int y, int z):B(x),D1(y),D2(z){std::cout<<"D3::D3 ("<<x<<","<<y<<","<<z<<" "<<std::endl;}
};

int main(){
    D3 d(4,5,6);
    std::cout<<d.get_a()<<std::endl;
    return 0;
}
```

```
B::B(4)
D1::D1(5)
D2::D2(6)
D3::D3(4, 5, 6)
4
```

Sfârșit capitol 3