



Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2021 – 2022
Semestrul II
Seriile 13, 14 și 15

Curs 6



Cuprins

Proiectarea descendentă a claselor. Moștenirea în C++.

- Controlul accesului la clasa de bază.
- Constructori, destructori și moștenire.
- Lista de initializare a constructorilor

Obs: în acest curs, exemplele vor fi luate, în principal, din cartea lui B. Eckel - Thinking in C++.



Moștenirea în C++

- important în C++ - reutilizare de cod;
- reutilizare de cod prin creare de noi clase (nu se dorește crearea de clase de la zero);
- 2 modalități (compunere și moștenire);
- “compunere” - noua clasă “este compusă” din obiecte reprezentând instanțe ale claselor deja create;
- “moștenire” - se creează un nou tip al unei clase deja existente.



Moștenirea în C++

Exemplu: compunere

```
class X { int i;  
public:  
    X() { i = 0; }  
    void set(int ii) { i = ii; }  
};  
class Y { int i;  
public:  
    X x; // Embedded object  
    Y() { i = 0; }  
    void f(int ii) { i = ii; }  
};  
int main() {  
    Y y;  
    y.f(47);  
    y.x.set(37); // Access the embedded object  
}
```



Moștenirea în C++

C++ permite moștenirea ceea ce înseamnă că putem deriva o clasă din altă clasă de bază sau din mai multe clase.

Prin derivare se obțin clase noi, numite clase derivate, care moștenesc proprietățile unei clase deja definite, numită clasă de bază.

Clasele derivate conțin toți membrii clasei de bază, la care se adaugă noi membri, date și funcții membre.

Dintr-o clasă de bază se poate deriva o clasă care, la rândul său, să servească drept clasă de bază pentru derivarea altora. Prin această succesiune se obține o **ierarhie de clase**.

Se pot defini clase derivate care au la bază mai multe clase, înglobând proprietățile tuturor claselor de bază, procedeu ce poartă denumirea de **moștenire multiplă**.



Moștenirea în C++

C++ permite moștenirea ceea ce înseamnă că putem deriva o clasă din altă clasă de bază sau din mai multe clase.

Sintaxa:

```
class Clasa_Derivata : [modificatori de acces] Clasa_de_Baza { .... } ;
```

sau

```
class Clasa_Derivata : [modificatori de acces] Clasa_de_Baza1, [modificatori  
de acces] Clasa_de_Baza2, [modificatori de acces] Clasa_de_Baza3 .....
```

Clasa de bază se mai numește clasă părinte sau superclasă, iar clasa derivată se mai numește subclasă sau clasă copil.



Moștenirea în C++

Exemplu: moștenire

```
class X { int i;
public:
    X() { i = 0; }
    void set(int ii) { i = ii; }
    int read() const { return i; }
    int permute() { return i = i * 47; }
};

class Y : public X {
    int i; // Different from X's I
public:
    Y() { i = 0; }
    int change() {
        i = permute(); // Different name call
        return i;
    }
    void set(int ii) {
        i = ii;
        sX::set(ii); // Same-name function call
    }
};
```

```
int main() {
    cout << sizeof(X) << sizeof(Y);

    Y D;
    D.change(); // X function interface comes through:
    D.read();
    D.permute(); // Redefined functions hide base versions:
    D.set(12);
}
```



Moștenirea în C++

Moștenire vs. Compunere

Moștenirea este asemănătoare cu procesul de includere a obiectelor în obiecte (procedeu ce poartă denumirea de compunere), dar există câteva elemente caracteristice moștenirii:

- codul poate fi comun mai multor clase;
- clasele pot fi extinse, fără a recompila clasele originare;
- funcțiile ce utilizează obiecte din clasa de bază pot utiliza și obiecte din clasele derivate din această clasă.



Moștenirea în C++

Modificatorii de acces la moștenire

```
class A : public B { /* declarații */};
```

```
class A : protected B { /* declarații */};
```

```
class A : private B { /* declarații */};
```

Dacă modificatorul de acces la moștenire este **public**, membrii din clasa de bază își păstrează tipul de acces și în derivată.

Dacă modificatorul de acces la moștenire este **private**, toți membrii din clasa de bază vor avea tipul de acces “private” în derivată, indiferent de tipul avut în bază.

Dacă modificatorul de acces la moștenire este **protected**, membrii “publici” din clasa de bază devin “protected” în clasa derivată, restul nu se modifică.



Moștenirea în C++

```
class Baza { int a;
```

```
public:
```

```
    void f() {cout<<a;} /// a este privat dar accesibil in clasa
```

```
private:
```

```
    void g() {cout<<a;}  
};
```

```
class Derivata1 : protected Baza{
```

```
public:
```

```
    void h() {cout<<a;} /// a este privat, inaccesibil  
};
```

```
class Derivata2: public Derivata1 {
```

```
public:
```

```
    void z() {cout<<a;}  
};
```

```
int main(){
```

```
    Baza ob1;
```

```
    /// cout<<ob1.a; /* a este privat deci am  
    acces la el doar din Baza */  
}
```



Moștenirea in C++

```
class Baza {  
public:    void f() {cout<<"B";} };
```

```
class Derivata : public Baza{ };
```

```
int main()  
{  
    Derivata ob1;  
    ob1.f();  
}
```

Obs. Funcția f() este accesibila din derivata;

-modificatorul de acces la moștenire este “public”, deci f() ramane “public” si in Derivata, deci este accesibil la apelul din main()

```
class Baza {  
public:    void f() {cout<<"B";} };
```

```
class Derivata : private sau protected Baza{ };
```

```
int main()  
{  
    Derivata ob1;  
    ob1.f(); // inaccesibil  
}
```

-Daca modificatorul de acces la moștenire este “private”, f() devine private in Derivata, deci inaccesibila in main.

-Daca modificatorul de acces la moștenire este “protected”, f() devine protected in Derivata, deci inaccesibil in main



Moștenirea în C++

Moștenirea cu specificatorul “public”

```
class Base {  
protected:  
    int i;  
public:  
    Base() : i(7) {}  
};
```

```
class Derived1 : public Base { };
```

```
class Derived2 : public Derived1 {  
public:  
    void set(int x) { i = x; }  
};
```

```
•int main() {  
    •    Derived2;  
    •    d.set(10);  
    •}
```

- dacă în baza avem zone “protected” ele sunt transmise și în derivata 1,2 tot ca protected, deci `i` e accesibil în funcția `set()`



Moștenirea în C++

Moștenirea cu specificatorul “private”

- inclusă în limbaj pentru completitudine;
- este mai bine a se utiliza compunerea în locul moștenirii private;
- toți membrii private din clasa de bază sunt ascunși în clasa derivată, deci inaccesibili;
- toți membrii public și protected devin private, dar sunt accesibile în clasa derivată;
- un obiect obținut printr-o astfel de derivare se tratează diferit față de cel din clasa de bază, e similar cu definirea unui obiect de tip bază în interiorul clasei noi (fără moștenire).
- dacă în clasa de bază o componentă era public, iar moștenirea se face cu specificatorul private, se poate reveni la public utilizând:

using Baza::nume_componenta



1. Moștenirea in C++

Moștenirea cu specificatorul “private”

```
class Base {
```

```
protected:
```

```
int i;
```

```
public:
```

```
Base() : i(7) {}
```

```
};
```

```
class Derived1 : private Base { };
```

```
class Derived2 : public Derived1 {
```

```
public:
```

```
void set(int x) { i = x; }
```

```
};
```

```
•int main() {
```

```
• Derived2;
```

```
• d.set(10);
```

```
• }
```

- moștenire derivata1 din baza (private)
atunci zonele protected devin private in
derivata1 si neaccesibile in derivata2.



Moștenirea in C++

Moștenirea cu specificatorul “private”

```
class Pet {  
public:  
    char eat() const { return 'a'; }  
    int speak() const { return 2; }  
    float sleep() const { return 3.0; }  
    float sleep( int) const { return 4.0; }  
};
```

```
class Goldfish : Pet { // Private inheritance  
public:  
    using Pet::eat; // Name publicizes member  
    using Pet::sleep; // Both overloaded members exposed  
};
```

```
int main() {  
    Goldfish bob;  
    bob.eat();  
    bob.sleep();  
    bob.sleep(1);  
    //! bob.speak();// Error: private member  
    function  
}
```



1. Moștenirea în C++

Moștenirea cu specificatorul “protected”

- secțiuni definite ca protected sunt similare ca definire cu private (sunt ascunse de restul programului), cu excepția claselor derivate;
- good practice: cel mai bine este ca variabilele de instanță să fie PRIVATE și funcții care le modifică să fie protected;
- Sintaxă: *class derivată: protected baza {...};*
- toți membrii publici și protected din baza devin protected în derivată;
- nu se prea folosește, inclusă în limbaj pentru completitudine.



1. Moștenirea în C++

Moștenirea cu specificatorul “protected”

```
class Base {  
protected:  
    int i;  
public:  
    Base() : i(7) {}  
};
```

```
class Derived1 : protected Base { };
```

```
class Derived2 : public Derived1 {  
public:  
    void set(int x) { i = x; }  
};
```

```
int main() {  
    Derived2 d;  
    d.set(10);  
}
```

- dacă în baza avem zone “protected” ele sunt transmise și în derivata 1,2 tot ca protected, deci `i` e accesibil în funcția `set()`



Moștenirea în C++

Inițializare de obiecte

Foarte important în C++: garantarea inițializării corecte => trebuie să fie asigurată și la compunere și moștenire.

La crearea unui obiect, compilatorul trebuie să garanteze apelul TUTUROR sub-obiectelor.

Problemă: - cazul sub-obiectelor care nu au constructori implicați sau schimbarea valorii unui argument default în constructor.

- Initializarea constantelor?

De ce? - constructorul noii clase nu are permisiunea să acceseze datele **private** ale sub-obiectelor, deci nu le pot inițializa direct.

Rezolvare: - o sintaxă specială: *lista de inițializare pentru constructori*.



Moștenirea în C++

Exemplu: lista de inițializare pentru constructori

```
class Bar {  
    int x;  
    public:  
        Bar(int i) {x = i;}  
};
```

```
class MyType: public Bar {  
    public:  
        MyType(int);  
};
```

```
MyType :: MyType (int i) : Bar (i) { ... }
```



Moștenirea în C++

Exemplu 2: lista de inițializare pentru constructori

```
class Alta_clasa { int a;  
    public:  
    Alta_clasa(int i) {a = i;}  
};
```

```
class Bar { int x;  
    public:  
    Bar(int i) {x = i;}  
};
```

```
class MyType2: public Bar {  
    Alta_clasa m; // obiect m = subobiect in cadrul clasei MyType2  
    public:  
    MyType2(int);  
};
```

```
MyType2 :: MyType2 (int i) : Bar (i), m(i+1) { ... }
```



Moștenirea în C++

Chestiune specială: “pseudo - constructori” pentru tipuri de bază

- membrii de tipuri predefinite nu au constructori;
- soluție: C++ permite tratarea tipurilor predefinite asemănător unei clase cu o singură dată membră și care are un constructor parametrizat.



Moștenirea în C++

```
class X {  
    int i;  
    float f;  
    char c;  
    char* s;  
public:  
    X() : i(7), f(1.4), c('x'), s("howdy") {}  
};  
  
int main() {  
    X x;  
    int i(100); // Applied to ordinary definition  
    int* ip = new int(47);  
}
```



Moștenirea în C++

Exemplu: compunere și moștenire

```
class A { int i;  
public:  
    A(int ii) : i(ii) {}  
    ~A() {}  
    void f() const {}  
};
```

```
class B { int i;  
public:  
    B(int ii) : i(ii) {}  
    ~B() {}  
    void f() const {}  
};
```

```
class C : public B {  
    A a;  
public:  
    C(int ii) : B(ii), a(ii) {}  
    ~C() {} // Calls ~A() and ~B()  
    void f() const  
        { // Redefinition  
            a.f();  
            B::f();  
        }  
};  
  
int main() {  
    C c(47);  
}
```



Moștenirea în C++

Constructorii clasei derivate

Pentru crearea unui obiect al unei clase derivate, **se creează inițial un obiect al clasei de bază prin apelul constructorului acesteia**, apoi se adaugă elementele specifice clasei derivate prin apelul constructorului clasei derivate.

Declarația obiectului derivat trebuie să conțină valorile de inițializare, **atât pentru elementele specifice, cât și pentru obiectul clasei de bază**.

Această specificare se atașează la antetul funcției constructor a clasei derivate.

În situația în care clasa de bază are definit un **constructor implicit** sau **constructor cu parametri implicați**, nu se impune specificarea parametrilor care se transferă către obiectul clasei de bază.



Moștenirea în C++

Constructorii clasei derivate

Constructorul parametrizat

```
class Forma {  
    protected:  
        int h;  
    public:  
        Forma(int a = 0) { h = a; }  
};
```

```
class Cerc: public Forma {  
    protected:  
        float raza;  
    public:  
        Cerc(int h=0, float r=0) : Forma(h) { raza = r; }  
};
```



Moștenirea în C++

Constructorii clasei derivate

Constructorul de copiere

Se pot distinge mai multe situații.

- 1) Dacă ambele clase, atât clasa derivată cât și clasa de bază, nu au definit un constructor de copiere, se apelează constructorul implicit creat de compilator. Copierea se face membru cu membru.
- 2) Dacă clasa de bază are constructorul de copiere definit, dar clasa derivată nu, pentru clasa derivată compilatorul creează un constructor implicit care apelează constructorul de copiere al clasei de bază.
- 3) Dacă se definește constructor de copiere pentru clasa derivată, acestuia **îi revine în totalitate sarcina transferării** valorilor corespunzătoare membrilor ce aparțin clasei de bază.



Moștenirea în C++

Constructorii clasei derivate

Constructorul de copiere

```
class Forma {  
protected:    int h;  
public:  
    Forma(const Forma& ob)    {    h = ob.h;    }  
};
```

```
class Cerc: public Forma {  
protected:  
    float raza;  
public:  
    Cerc(const Cerc&ob):Forma(ob)    {    raza = ob.raza;    }  
};
```



Moștenirea în C++

Ordinea apelării constructorilor și destructorilor

- constructorii sunt apelați în ordinea definirii obiectelor ca membri ai clasei și în ordinea moștenirii:
- la fiecare nivel se apelează:
 - **întâi constructorul de la moștenire,**
 - apoi **constructorii din obiectele membru** în clasa respectivă (care sunt apelați în ordinea definirii)
 - și la final se merge pe următorul nivel în ordinea moștenirii;
- destructorii sunt executați în ordinea inversă a constructorilor



Moștenirea în C++

Ordinea apelării constructorilor și destructorilor

```
class A {  
public:  
    A(){cout<<"A ";}  
    ~A(){cout<<"~A ";}  
};
```

```
class C {  
public:  
    C(){cout<<"C ";}  
    ~C(){cout<<"~C ";}  
};
```

Ordine: C B A D ~D ~A ~B ~C

```
class B {  
    C ob;  
public:  
    B(){cout<<"B ";}  
    ~B(){cout<<"~B ";}  
};  
class D: public B {  
    A ob;  
public:  
    D(){cout<<"D ";}  
    ~D(){cout<<"~D ";}  
};  
int main() {  
    D s;  
}
```



Moștenirea in C++

Ordinea chemării constructorilor și destructorilor

```
#define CLASS(ID) class ID { \
public: \
    ID(int) \
        { cout << #ID " constructor\n"; } \
    ~ID() \
        { cout << #ID " destructor\n"; } \
};
```

```
CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);
```

```
class Derived1 : public Base1 {
    Member1 m1;
    Member2 m2;
public:
    Derived1(int) : m2(1), m1(2), Base1(3)
        { cout << "Derived1 constructor\n"; }
    ~Derived1() { cout << "Derived1 destructor\n"; }
};
```

```
class Derived2 : public Derived1 {
    Member3 m3;
    Member4 m4;
public:
    Derived2() : m3(1), Derived1(2), m4(3)
        { cout << "Derived2 constructor\n"; }
    ~Derived2() { cout << "Derived2 destructor\n"; }
};
```

```
int main() { Derived2 d2; }
```



Moștenirea in C++

Ordinea chemării constructorilor și destructorilor

```
#define CLASS(ID) class ID { \
public: \
    ID(int)
        { cout << #ID " constructor\n"; } \
    ~ID()
        { cout << #ID " destructor\n"; } \
};
```

```
CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);
```

Se va afisa:

Base1 constructor
Member1 constructor
Member2 constructor
Derived1 constructor
Member3 constructor
Member4 constructor
Derived2 constructor
Derived2 destructor
Member4 destructor
Member3 destructor
Derived1 destructor
Member2 destructor
Member1 destructor
Base1 destructor



Moștenirea în C++

Operatorul=

```
class Forma {  
protected:  
    int h;  
public:  
    Forma& operator=(const Forma& );  
};  
class Cerc: public Forma {  
protected:  
    float raza;  
public:  
    Cerc& operator=(const Cerc& );  
};
```

```
Forma& Forma::operator=(const Forma& ob) {  
    if (this!=&ob) { h = ob.h; }  
    return *this;  
}  
  
Cerc& Cerc::operator= (const Cerc& ob) {  
    if (this != &ob)  
    {  
        this->Forma::operator=(ob);  
    }  
    return *this;  
}
```




Moștenirea în C++

Redefinirea funcțiilor membre

```
class Forma {  
    protected:  
        int h;  
    public:  
        void afis() { cout<<h<<" "; }  
};
```

Clasa derivată are acces la toți membrii cu acces **protected** sau **public** ai clasei de bază.

```
class Cerc: public Forma {  
    protected:  
        float raza;  
    public:  
        void afis() {  
            Forma::afis();  
            cout<<raza; }  
};
```

Este permisă **supradefinirea** funcțiilor membre clasei de bază cu funcții membre ale clasei derivate.



Moștenirea în C++

Compatibilitatea între o clasă derivată și clasa de bază. Conversii de tip

Deoarece clasa derivată moștenește proprietățile clasei de bază, între tipul clasă derivată și tipul clasă de bază se admite o anumită compatibilitate.

Compatibilitatea este valabilă numai pentru clase **derivate cu acces public** la clasa de bază și numai în sensul de la clasa derivată spre cea de bază, nu și invers.

Compatibilitatea se manifestă sub forma unor **conversii implicite de tip**:

- dintr-un obiect derivat într-un obiect de bază;
- dintr-un pointer sau referință la un obiect din clasa derivată într-un pointer sau referință la un obiect al clasei de bază.



Perspective

Cursul 7:

1. Proiectarea descendenta a claselor.
 - Redefinirea membrilor unei clase de bază într-o clasă derivată.
 - Mostenirea si functiile statice
 - Declarații de acces.

2. Parametrizarea metodelor (polimorfism la execuție).
 - Funcții virtuale în C++. Clase abstracte.

 - Destructori virtuali.