



Programare orientată pe obiecte

- suport de curs -

Anca Dobrovăț
Andrei Păun

An universitar 2022 – 2023
Semestrul I
Seriile 21 și 26

Curs 6

Agenda cursului

1. Proiectarea descendentă a claselor. Moștenirea în C++ (recapitulare și completări la cursul 5)

Modificatori de acces, constructori / destructori, redefinirea membrilor unei clase de bază într-o clasă derivată.

Mostenire din baza multipla. Mostenire in diamant

2. Polimorfism la execuție prin funcții virtuale în C++.

Parametrizarea metodelor (polimorfism la execuție).

Funcții virtuale în C++.

Clase abstracte.

Overloading pe funcții virtuale

Destructori și virtualizare

1. Moștenirea in 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_Derivată : [modificatori de acces] Clasa_de_Bază { } ;

sau

*class Clasa_Derivată : [modificatori de acces] Clasa_de_Bază1,
[modificatori de acces] Clasa_de_Bază2, [modificatori de acces]
Clasa_de_Bază3*

Clasă de bază se mai numește **clasa părinte** sau **superclasă**, iar clasa derivată se mai numește **subclasa** sau **clasa copil**.

1. Moștenirea in 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; X::set(ii); // Same-name function call
    }
};
```

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

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

1. Moștenirea în C++

Inițializare de obiecte

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

La crearea unui obiect, compilatorul trebuie să garanteze apelul TUTUROR subobiectelor.

Problema: - cazul subobiectelor care nu au constructori implicați sau schimbarea valorii unui argument default în constructor.

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

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

1. Moștenirea in C++

Exemple: 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) { ... }
```

1. Moștenirea in C++

Exemple: compoziție 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);  
}
```

1. Moștenirea in 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 clasele de bază au definit **constructor implicit** sau **constructor cu parametri implicați**, nu se impune specificarea parametrilor care se transferă către obiectul clasei de bază.

1. Moștenirea in 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 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. (poate fi considerata un caz particular al primei situații, deoarece și partea de bază poate fi privită ca un fel de membru, iar la copiere se apelează cc pentru fiecare membru).
- 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.

1. Moștenirea în C++

Ordinea chemării constructorilor și destructorilor

Constructorii sunt chemați 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 chemați în ordinea definirii) și la final constructorul propriu;
- se merge pe următorul nivel în ordinea moștenirii;

Destructorii sunt chemați în ordinea inversă a constructorilor

1. Moștenirea in C++

Redefinirea funcțiilor membre

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

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

-2 modalități de a redefini o funcție membră:

- **cu același antet ca în clasa de bază** (“redefining” - în cazul funcțiilor oarecare / “overloading” - în cazul funcțiilor virtuale);
- **cu schimbarea listei de argumente sau a tipului returnat.**

1. Moștenirea in C++

Redefinirea funcțiilor membre

Exemplu: - pastrarea antetului/tipului returnat

```
class Baza {  
public:  
    void afis() { cout<<"Baza\n"; }  
};
```

```
class Derivata : public Baza {  
public:  
    void afis() { Baza::afis(); cout<<"si Derivata\n"; }  
};
```

```
int main() {  
    Derivata d;  
    d.afis(); // se afiseaza "Baza si Derivata"  
}
```

1. Moștenirea in C++

Redefinirea funcțiilor membre

Exemplu: - nepastrarea antetului/tipului returnat

```
class Baza {  
public:  
    void afis() { cout<<"Baza\n"; }  
};
```

```
class Derivata : public Baza {  
public:  
    void afis (int x) {  
        Baza::afis();  
        cout<<"si Derivata\n"; }  
};
```

```
int main() {  
    Derivata d;  
    d.afis(); //nu exista Derivata::afis( )  
    d.afis(3); }
```

Obs: la redefinirea unei funcții din clasa de baza, toate celelalte versiuni sunt automat ascunse!

1. Moștenirea in C++

Redefinirea funcțiilor membre

Obs:

Schimbarea interfeței clasei de bază prin modificarea tipului returnat sau a semnăturii unei funcții, înseamnă, de fapt, utilizarea clasei în alt mod.

Scopul principal al moștenirii: polimorfismul.

Schimbarea semnăturii sau a tipului returnat = schimbarea interfeței = contravine exact polimorfismului (un aspect esențial este păstrarea interfeței clasei de bază).

1. Moștenirea in C++

Moștenirea si funcțiile statice

Funcțiile membre statice se comportă exact ca și funcțiile nemembre:
Se moștenesc în clasa derivată.

Redefinirea unei funcții membre statice duce la ascunderea celorlalte
supraîncărcări.

Schimbarea semnăturii unei funcții din clasa de bază duce la ascunderea
celorlalte versiuni ale funcției.

Dar: O funcție membră statică nu poate fi virtuală.

1. Moștenirea in 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.

1. 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 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 in C++

Moștenire multiplă (MM)

- putine limbaje au MM;
- moștenirea multiplă e complicată: ambiguitate LA MOSTENIREA IN ROMB / IN DIAMANT;
- nu e nevoie de MM (se simulează cu moștenire simplă);
- se moșteneste in același timp din mai multe clase;

Sintaxa:

*class Clasa_Derivată : [modificatori de acces] Clasa_de_Bază1,
[modificatori de acces] Clasa_de_Bază2, [modificatori de acces]
Clasa_de_Bază3*

1. Moștenirea in C++

Moștenire multiplă (MM)

Exemplu:

```
class Imprimanta { };  
class Scanner { };  
class Multifuncționala: public Imprimanta, public Scanner { };
```

Ce ar putea crea probleme in cazul urmator?

```
class Baza{ };  
class Derivata_1: public Baza{ };  
class Derivata_2: public Baza{ };  
class Derivata_3: public Derivata_1, public Derivata_2 { };
```

In **Derivata_3** avem de doua ori variabilele din **Baza**!!

1. Moștenirea in C++

Moștenire multiplă : ambiguitati (problema diamantului)

```
class base { public:    int i; };  
class derived1 : public base { public:    int j; };  
class derived2 : public base { public:    int k; };  
class derived3 : public derived1, public derived2 {public:    int sum; };
```

```
int main() {  
    derived3 ob;  
    ob.i = 10; // this is ambiguous, which i      // expl ob.derived1::i  
    ob.j = 20;  
    ob.k = 30;  
    ob.sum = ob.i + ob.j + ob.k; // i ambiguous here, too  
    cout << ob.i << " "; // also ambiguous, which i?  
    cout << ob.j << " " << ob.k << " ";  
    cout << ob.sum;  
    return 0;  
}
```

1. Moștenirea in C++

Moștenire multiplă (MM)

- dar dacă avem nevoie doar de o copie lui i?
- nu vrem să consumăm spațiu în memorie;
- *folosim moștenire virtuală:*

```
class base { public:    int i; };  
class derived1 : virtual public base { public:    int j; };  
class derived2 : virtual public base { public:    int k; };  
class derived3 : public derived1, public derived2 {public:    int sum; };
```

- Dacă avem moștenire de două sau mai multe ori dintr-o clasă de bază (fiecare moștenire trebuie să fie virtuală) atunci compilatorul alocă spațiu pentru o singură copie;
- În clasele derived1 și 2 moștenirea e la fel ca mai înainte (niciun efect pentru virtual în acel caz)

2. Polimorfismul la execuție prin funcții virtuale

Funcții virtuale

Funcțiile virtuale și felul lor de folosire: componentă IMPORTANTĂ a limbajului OOP.

Folosit pentru polimorfism la execuție ---> cod mai bine organizat cu polimorfism.

Codul poate “crește” fără schimbări semnificative: programe extensibile.

Funcțiile virtuale sunt definite în bază și redefinite în clasa derivată.

Pointer de tip bază care arată către obiect de tip derivat și cheamă o funcție virtuală în bază și redefinite în clasa derivată executa ***Funcția din clasa derivată.***

Poate fi vazuta ca exemplu de separare dintre interfata si implementare.

2. Polimorfismul la execuție prin funcții virtuale

Decuplare în privința tipurilor

Upcasting - Tipul derivat poate lua locul tipului de bază (foarte important pentru procesarea mai multor tipuri prin același cod).

Funcții virtuale: ne lasă să chemăm funcțiile pentru tipul derivat.

Problemă: apel la funcție prin pointer (tipul pointerului ne da funcția apelată).

2. Polimorfismul la execuție prin funcții virtuale

```
enum note { middleC, Csharp, Eflat }; // Etc.
```

```
class Instrument { public:  
    void play(note) const {  
        cout << "Instrument::play" << endl; }  
};
```

```
class Wind : public Instrument {  
public: // Redefine interface function:  
    void play(note) const {  
        cout << "Wind::play" << endl; }  
};
```

```
void tune(Instrument& i) { i.play(middleC); }
```

```
int main() {  
    Wind flute;  
    tune(flute); // Upcasting ==> se afiseaza Instrument::play  
}
```

2. Polimorfismul la execuție prin funcții virtuale

In C ---> early binding la apel de funcții - se face la compilare.

In C++ ---> putem defini late binding prin funcții virtuale (late, dynamic, runtime binding) - se face apel de funcție bazat pe tipul obiectului, la rulare (nu se poate face la compilare).

Late binding ==> prin pointeri!

Late binding pentru o funcție: se scrie virtual înainte de definirea funcției.

Pentru clasa de bază: nu se schimbă nimic!

Pentru clasa derivată: late binding înseamnă că un obiect derivat folosit în locul obiectului de bază își va folosi funcția sa, nu cea din bază (din cauză de late binding).

Utilitate: putem extinde codul precedent fara schimbări in codul deja scris.

2. Polimorfismul la execuție prin funcții virtuale

```
enum note { middleC, Csharp, Eflat }; // Etc.
```

```
class Instrument { public:  
    virtual void play(note) const {  
        cout << "Instrument::play" << endl; }  
};
```

```
class Wind : public Instrument {  
public: // Redefine interface funcțion:  
    void play(note) const {  
        cout << "Wind::play" << endl; }  
};
```

```
void tune(Instrument& i) { i.play(middleC); }
```

```
int main() {  
    Wind flute;  
    tune(flute); // se afiseaza Wind::play  
}
```

2. Polimorfismul la execuție prin funcții virtuale

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.
```

```
class Instrument {
public:
    virtual void play(note) const {
        cout << "Instrument::play" << endl;
    }
};
```

```
// Wind objects are Instruments
// because they have the same interface:
```

```
class Wind : public Instrument {
public:
    // Override interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};
```

```
void tune(Instrument& i) {
    // ...
    i.play(middleC);
}
```

```
class Percussion : public Instrument {
public:
    void play(note) const {
        cout << "Percussion::play" << endl; } };
class Stringed : public Instrument {
public:
    void play(note) const {
        cout << "Stringed::play" << endl; } };
class Brass : public Wind {
public:
    void play(note) const {
        cout << "Brass::play" << endl; } };
class Woodwind : public Wind {
public:
    void play(note) const {
        cout << "Woodwind::play" << endl; } };
int main() {
    Wind flute;
    Percussion drum;
    Stringed violin;
    Brass flugelhorn;
    Woodwind recorder;
    tune(flute); tune(flugehorn); tune(violin);
}
```

2. Polimorfismul la execuție prin funcții virtuale

Cum se face late binding

Tipul obiectului este ținut în obiect pentru clasele cu funcții virtuale.

Late binding se face (uzual) cu o tabelă de pointeri: vptr către funcții.

În tabelă sunt adresele funcțiilor clasei respective (funcțiile virtuale sunt din clasa, celelalte pot fi moștenite, etc.).

Fiecare obiect din clasă are pointerul acesta în componență.

La apel de funcție membru se merge la obiect, se apelează funcția prin vptr.

Vptr este inițializat în constructor (automat).

2. Polimorfismul la execuție prin funcții virtuale

Cum se face late binding

```
class NoVirtual { int a;  
public:  
    void x() const {}  
    int i() const { return 1; } };
```

```
class OneVirtual { int a;  
public:  
    virtual void x() const {}  
    int i() const { return 1; } };
```

```
class TwoVirtuals { int a;  
public:  
    virtual void x() const {}  
    virtual int i() const { return 1; } };
```

```
int main() {  
    cout << "int: " << sizeof(int) << endl;  
    cout << "NoVirtual: "  
        << sizeof(NoVirtual) << endl;  
    cout << "void* : " << sizeof(void*) << endl;  
    cout << "OneVirtual: "  
        << sizeof(OneVirtual) << endl;  
    cout << "TwoVirtuals: "  
        << sizeof(TwoVirtuals) << endl;  
}
```

2. Polimorfismul la execuție prin funcții virtuale

Cum se face late binding

```
class Pet { public:  
    virtual string speak() const { return " "; } };  
  
class Dog : public Pet { public:  
    string speak() const { return "Bark!"; } };  
  
int main() {  
    Dog ralph;  
    Pet* p1 = &ralph;  
    Pet& p2 = ralph;  
    Pet p3;  
    // Late binding for both:  
    cout << "p1->speak() = " << p1->speak() << endl;  
    cout << "p2.speak() = " << p2.speak() << endl;  
    // Early binding (probably):  
    cout << "p3.speak() = " << p3.speak() << endl;  
}
```


2. Polimorfismul la execuție prin funcții virtuale

Daca funcțiile virtuale sunt așa de importante de ce nu sunt toate funcțiile definite virtuale (din oficiu)?

Deoarece “costă” în viteza programului.

În Java sunt “default”, dar Java ocupa mai multa memorie.

Nu mai putem avea funcții inline (ne trebuie adresa funcției pentru **vp_{tr}**).

2. Polimorfismul la execuție prin funcții virtuale

Clase abstracte și funcții virtuale pure

Clasă abstractă = clasă care are cel puțin o funcție virtuală PURĂ

Necesitate: clase care dau doar interfață (nu vrem obiecte din clasă abstractă ci upcasting la ea).

Eroare la instantierea unei clase abstracte (nu se pot defini obiecte de tipul respectiv).

Permisă utilizarea de pointeri și referințe către clasă abstractă (pentru upcasting).

Nu pot fi trimise către funcții (prin valoare).

2. Polimorfismul la execuție prin funcții virtuale

Funcții virtuale pure

Sintaxa: **virtual** *tip_returnat nume_funcție(lista_parametri) =0;*

Ex: virtual int pura(int i)=0;

Obs: La moștenire, dacă în clasa derivată nu se definește funcția pură, clasa derivată este și ea clasă abstractă ---> nu trebuie definită funcție care nu se execută niciodată

UTILIZARE IMPORTANTĂ: prevenirea “object slicing”.

2. Polimorfismul la execuție prin funcții virtuale

Clase abstracte si funcții virtuale pure

```
class Pet { string pname;  
public:  
    Pet(const string& name) : pname(name) {}  
    virtual string name() const { return pname; }  
    virtual string description() const {  
        return "This is " + pname;  
    }  
};  
  
class Dog : public Pet { string favoriteActivity;  
public:  
    Dog(const string& name, const string& activity)  
        : Pet(name), favoriteActivity(activity) {}  
    string description() const {  
        return Pet::name() + " likes to " + favoriteActivity;  
    }  
};
```

```
void describe(Pet p) { // Slicing  
    cout << p.description() << endl;  
}  
  
int main() {  
    Pet p("Alfred");  
    Dog d("Fluffy", "sleep");  
    describe(p);  
    describe(d);  
}
```

2. Polimorfismul la execuție prin funcții virtuale

Overload pe funcții virtuale

Obs. Nu e posibil overload prin schimbarea tipului param. de întoarcere (e posibil pentru ne-virtuale)

De ce. Pentru că se vrea să se garanteze că se poate chema baza prin apelul respectiv.

Excepție: pointer către bază întors în bază, pointer către derivată în derivată

2. Polimorfismul la execuție prin funcții virtuale

Overload pe funcții virtuale

```
class Base {  
public:  
    virtual int f() const {  
        cout << "Base::f()\n"; return 1; }  
    virtual void f(string) const {}  
    virtual void g() const {}  
};  
  
class Derived1 : public Base {public:  
    void g() const {}  
};  
  
class Derived2 : public Base {public:  
    // Overriding a virtual function:  
    int f() const { cout << "Derived2::f()\n";  
        return 2; }  
};
```

```
int main() {  
    string s("hello");  
    Derived1 d1;  
    int x = d1.f();  
    d1.f(s);  
    Derived2 d2;  
    x = d2.f();  
    //! d2.f(s); // string version hidden  
}
```

2. Polimorfismul la execuție prin funcții virtuale

Overload pe funcții virtuale

```
class Base {  
public:  
    virtual int f() const {  
        cout << "Base::f()\n"; return 1; }  
    virtual void f(string) const {}  
    virtual void g() const {}  
};  
  
class Derived3 : public Base {public:  
    //! void f() const{ cout << "Derived3::f()\n";}};  
  
class Derived4 : public Base {public:  
    // Change argument list:  
    int f(int) const  
        { cout << "Derived4::f()\n"; return 4; }  
};
```

```
int main() {  
    string s("hello");  
    Derived4 d4;  
    x = d4.f(1);  
    //! x = d4.f(); // f() version hidden  
    //! d4.f(s); // string version hidden  
    Base& br = d4; // Upcast  
    //! br.f(1); // Derived version  
    //! br.f(s); // Base version available  
    return 0;  
}
```

2. Polimorfismul la execuție prin funcții virtuale

Constructori si virtualizare

Obs. NU putem avea constructori virtuali.

În general pentru funcțiile virtuale se utilizează late binding, dar în utilizarea funcțiilor virtuale în constructori, varianta locală este folosită (early binding)

De ce?

Pentru că funcția virtuală din clasa derivată ar putea crede că obiectul e inițializat deja

Pentru că la nivel de compilator în acel moment doar VPTR local este cunoscut

2. Polimorfismul la execuție prin funcții virtuale

Destructori si virtualizare

Este uzual să se întâlnească.

Se cheamă în ordine inversă decât constructorii.

Dacă vrem să eliminăm porțiuni alocate dinamic și pentru clasa derivată dar facem upcasting trebuie să folosim destructori virtuali.

2. Polimorfismul la execuție prin funcții virtuale

Destructorii și virtualizare

```
class Base1 {public: ~Base1() { cout << "~Base1()\n"; } };
```

```
class Derived1 : public Base1 {public: ~Derived1() { cout << "~Derived1()\n"; } };
```

```
class Base2 {public:  
    virtual ~Base2() { cout << "~Base2()\n"; }  
};
```

```
class Derived2 : public Base2 {public: ~Derived2() { cout << "~Derived2()\n"; } };
```

```
int main() {  
    Base1* bp = new Derived1;  
    delete bp; // Afis: ~Base1()  
    Base2* b2p = new Derived2;  
    delete b2p; // Afis: ~Derived2() ~Base2()  
}
```

2. Polimorfismul la execuție prin funcții virtuale

Destructorii virtuali puri

Utilizare: recomandat să fie utilizat dacă mai sunt și alte funcții virtuale.

Restricție: trebuie să fie definiți în clasă (chiar dacă este abstractă).

La moștenire nu mai trebuie să fie redefiniți (se construiește un destructor din oficiu)

De ce? Pentru a preveni instantierea clasei.

Obs. Nu are nici un efect dacă nu se face upcasting.

```
class AbstractBase {
```

```
public:
```

```
virtual ~AbstractBase() = 0;
```

```
};
```

```
AbstractBase::~~AbstractBase() {}
```

```
class Derived : public AbstractBase {};
```

```
// No overriding of destructor necessary?
```

```
int main() { Derived d; }
```

2. Polimorfismul la execuție prin funcții virtuale

Funcții virtuale in destructori

La apel de funcție virtuală din funcții normale se apelează conform VPTR

În destructori se face early binding! (apeluri locale)

De ce? Pentru că acel apel poate să se bazeze pe porțiuni deja distruse din obiect

```
class Base { public:  
    virtual ~Base() { cout << "~Base1()\n"; this->f(); }  
    virtual void f() { cout << "Base::f()\n"; }  
};  
class Derived : public Base { public:  
    ~Derived() { cout << "~Derived()\n"; }  
    void f() { cout << "Derived::f()\n"; }  
};  
  
int main() {  
    Base* bp = new Derived;  
    delete bp; // Afis: ~Derived() ~Base1() Base::f()  
}
```

2. Polimorfismul la execuție prin funcții virtuale

Downcasting

Folosit in ierarhii polimorifice (cu funcții virtuale).

Problema: upcasting e sigur pentru că respectivele funcții trebuie să fie definite în bază, downcasting e problematic.

Explicit cast prin: **dynamic_cast**

Dacă știm cu siguranță tipul obiectului putem folosi “static_cast”.

Static_cast întoarce pointer către obiectul care satisface cerințele sau 0.

Folosește tabelele VTABLE pentru determinarea tipului.

2. Polimorfismul la execuție prin funcții virtuale

Downcasting

```
class Pet { public: virtual ~Pet(){};
class Dog : public Pet {};
class Cat : public Pet {};
```

```
int main() {
    Pet* b = new Cat; // Upcast
    Dog* d1 = dynamic_cast<Dog*>(b); // Afis - 0; Try to cast it to Dog*:
    Cat* d2 = dynamic_cast<Cat*>(b); // Try to cast it to Cat*:
    // b si d2 retin aceeași adresa
    cout << "d1 = " << d1 << endl;
    cout << "d2 = " << d2 << endl;
    cout << "b = " << b << endl;
}
```

2. Polimorfismul la execuție prin funcții virtuale

Downcasting

```
class Shape { public: virtual ~Shape() {} };  
class Circle : public Shape {};  
class Square : public Shape {};  
class Other {};
```

```
int main() {  
    Circle c;  
    Shape* s = &c; // Upcast: normal and OK  
    // More explicit but unnecessary:  
    s = static_cast<Shape*>(&c);  
    // (Since upcasting is such a safe and common  
    // operation, the cast becomes cluttering)  
    Circle* cp = 0;  
    Square* sp = 0;
```

// Static Navigation of class hierarchies
requires extra type information:

```
    if(typeid(s) == typeid(cp)) // C++ RTTI  
        cp = static_cast<Circle*>(s);  
    if(typeid(s) == typeid(sp))  
        sp = static_cast<Square*>(s);  
    if(cp != 0)  
        cout << "It's a circle!" << endl;  
    if(sp != 0)  
        cout << "It's a square!" << endl;  
    // Static navigation is ONLY an efficiency  
    hack;  
    // dynamic_cast is always safer. However:  
    // Other* op = static_cast<Other*>(s);  
    // Conveniently gives an error message,  
    while  
        Other* op2 = (Other*)s;  
    // does not  
}
```

Perspective

Cursul 7:

Mostenire – completari:

Copy constructor, operator =, destructor pt clasele cu attribute de tip pointer

- clone, copy&swap,
- RAI (*Resource Acquisition Is Initialization*)

Tratarea exceptiilor