

# POO. Curs 8

Anca Dobrovăț și Andrei Păun. Document editat de Raluca Tudor

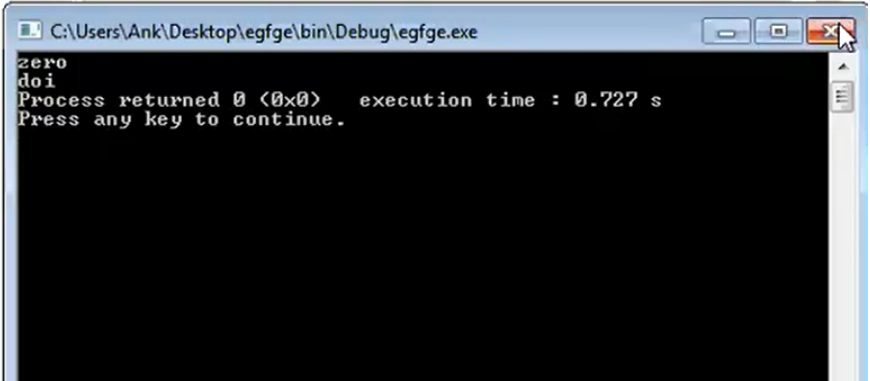
8 Aprilie 2020

## 1 Proiectarea descendentă a claselor. Moștenirea în C++

### 1.1 Constructorii clasei derivate

```
1 #include <iostream>
2 #include <typeinfo>
3
4 using namespace std;
5
6 class B
7 {
8 public:
9     B() {cout<<"zero"<<endl;}
10    B(int x){cout<<"unu"<<endl;}
11 };
12
13 class D: public B
14 {
15 public:
16     D(int x, int y){cout<<"doi"};
17 };
18
19 int main()
20 {
21     D ob(3,4);
22 }
23
```

SE FACE TRIMITERE CĂTRE CONSTRUCTORUL DE INIȚIALIZARE DIN CLASA DE BAZĂ

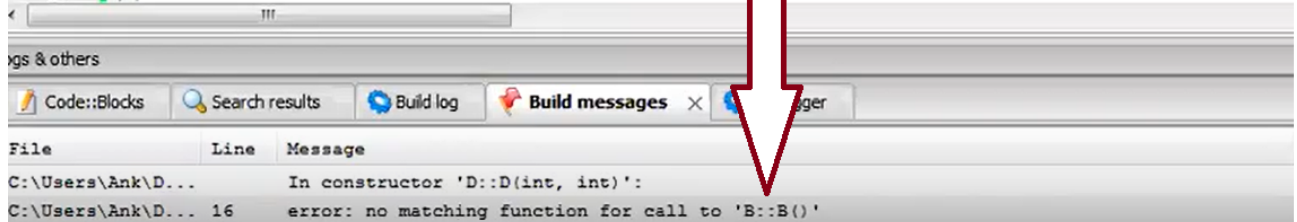


În schimb, dacă avem:

```
5
6 class B
7 {
8 public:
9     // B() {cout<<"zero"<<endl;}
10    B(int x){cout<<"unu"<<endl;}
11 };
12
13 class D: public B
14 {
15 public:
16     D(int x, int y){cout<<"doi"};
17 };
18
19 int main()
20 {
21     D ob(3,4);
22 }
23
24 /*
25 /// modifcatori de acces la mostenire
26 class B
27 {

```

EROARE CĂ NU GĂSEȘTE CONSTRUCTORUL DE INIȚIALIZARE



Regula este ca atunci când se creează un obiect derivat, constructorii lui să facă apel implicit către constructorul de inițializare din clasa de bază.

Același lucru se întâmplă și pentru constructorul de copiere. (vezi curs 6)

## 1.2 Modificatorii de acces la moștenire

```
25 // modificatori de acces la mostenire
26 class B
27 {
28     int a;
29     protected: int b;
30     public: int c;
31 };
32
33 class D:B // modificador default privat
34 {
35     public:
36     void f()
37     {
38         //cout<<a<<endl; ///inaccesibil
39         cout<<b<<endl;
40         cout<<c<<endl;
41         D ob;
42         //cout<<ob.a<<endl; ///inaccesibil
43         cout<<ob.b<<endl;
44         cout<<ob.c<<endl;
45         B ob2;
46         //cout<<ob2.a<<endl; ///inaccesibil
47         //cout<<ob2.b<<endl; ///protected in this context !!!!
48         cout<<ob2.c<<endl;
49     }
50 };
```

### 1.3 Ordinea constructorilor

```
62  ///constructori clasa derivata
63  class A
64  {
65  public:
66      A(){cout<<"A";}
67  };
68
69  class B
70  {
71  public:
72      B(){cout<<"B";}
73  };
74
75  class D: public B
76  { A ob;
77  public:
78      D(){cout<<"D";}
79  };
80
81  int main()
82  {
83      D ob; //ordine constructori BAD - BAZA / OBIECTE CONTINUTE / DERIVATA
84  }
```

### 1.4 Transmiterea parametrilor către constructorul din clasa de bază

```
88  /// transmiterea parametrilor catre constructorul din clasa de baza
89
90  class B
91  {
92  protected
93  int x, z;
94  public:
95      B(int a = 70, int b = 80){x = a;}
96      B(const B& ob){x = ob.x;}
97      void f(){cout<<x<<endl;}
98  };
99
100  class D: public B
101  {
102  int y;
103  public:
104      D(int a = 70, int b = 80, int c = 90) B(a,b) y = c;
105      D(const D& ob) B(ob) y = ob.y; //apel explicit constructor copiere din baza :B(ob)
106      void f(){cout<<x<<" " <<y<<endl;}
107  };
108
109  int main()
110  {
111      D ob(4,5);
112      ob.f();
113      D ob2 = ob;
114      ob2.f();
115  }
```

## 1.5 Operatorii se moștenesc?

Da! Dar operatorul de atribuire NU! Pe el îl tratăm asemănător cu un constructor!

```
118 // operatorii se moștenesc?
119
120 class B
121 {
122     protected:
123         int x;
124     public:
125         B(){}
126         B(int a){x = a;}
127         void f(){cout<<x<<endl;}
128         B operator-(){x = -x;}
129         friend istream& operator>>(istream& in, B& ob){in>>ob.x; return in;}
130 };
131
132 class D: public B
133 {
134     int y;
135     public:
136         D(){}
137         D(int a, int b):B(a){y = b;}
138         void f(){ B::f(); cout<<y<<endl;}
139         D operator-(){x = -x; y = -y;}
140         friend istream& operator>>(istream& in, D& ob){
141             in>>(B&)ob; //aici avem conversia din C... in viitor vom folosi mai mult dynamic_cast
142             in>>ob.y; return in;}
143 };
```

## 1.6 Moștenire multiplă

```
182 // mostenire multipla
183 class A
184 {
185     protected:
186         int x;
187 };
188
189 class B
190 {
191     protected:
192         int x;
193 };
194
195 class C: public A, public B
196 {
197     public:
198         //void f(){cout<<x;} ///ambiguu care x?
199         void f(){cout<<A::x;} //explicit care x, din ce clasa
200 };
201
202 int main()
203 {
204     C ob;
205     ob.f();
206 }
```

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

### 2.1 Ierarhii polimorifice/ nepolimorifice

Când nu am cuvântul cheie virtual în funcțiile din clasele de bază, de exemplu, care sunt redefinite la moștenire, și nu am nici pointeri -> totul este definit la compilare, deci se cunoaște de la început ce tip ce date am. Dacă am cuvântul cheie virtual în funcții, dar nu folosesc pointeri/ referințe -> iarăși tipurile sunt definite la compilare și nu la execuție. Dacă am și funcții virtuale, și pointeri (în sensul că iau un pointer către clasa de bază care să rețină adresa derivatei) -> tipul declarat <> tipul real.

În următorul exemplu, nu folosesc virtual, deci tipul este definit la compilare.

```
236 // ierarhie nepolimorfica ==> decizie tipuri la compilare ==> early binding
237 class B{
238 public:
239     void f(){cout<<"B\n";}
240 };
241
242 class D: public B
243 {
244 public:
245     void f(){cout<<"D\n";}
246 };
247
248 int main()
249 {
250     B ob1;
251     ob1.f();
252     D ob2;
253     ob2.f();
254     B* p = &ob2; /// upcasting
255     p->f();
256 }
```

Atenție

```
248 int main()
249 {
250     B ob1;
251     ob1.f();
252     D ob2;
253     ob2.f();
254     B* p = &ob2; /// upcasting
255     p->D::f();
256 }
```

Dacă adaug cuvântul cheie virtual, atunci trec ierarhia dintr-una nepolimorfică într-una polimorfică și se va afișa B D D.

```

366 class B{
367 public:
368     virtual void f(){cout<<"B\n";}
369 };
370
371 class D: public B
372 {
373 public:
374     void f(){cout<<"D\n";}
375     virtual void g(){cout<<"gD\n";}
376 };
377
378 class D2: public D
379 {
380 public:
381     //void f(){cout<<"D2\n";}
382 };
383
384 int main()
385 {
386     D2 ob;
387     ob.f();
388     B *p = &ob; Atenție: dacă apelez funcția f pentru p, atunci se va apela f-ul din D, nu din B
389     p->g(); // error "class B" has no member named g
390 }

```

**DEOARECE AM IERARHIE POLIMORFICĂ!**

**Atenție!!!** Nu pot apela funcția g pentru p pentru că nu o recunoaște, deoarece pointerul p este la origine un pointer către clasa de bază! Deci acest late binding are o limită! Nu pot să folosesc ceea ce nu știu! În cazul nostru, g-ul a apărut în vtable mai târziu!

## 2.2 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) -> *deoarece pointerii și referințele lucrează cu nr. întregi, cu adrese, nu cu obiecte.*

Nu pot fi trimise către funcții (prin valoare).

### Funcții virtuale pure

Sintaxa:

```

1 virtual tip_returnat nume_functie(lista_parametri) =0;
2
3 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".

```

395  /// Exemplificare clasa abstracta
396  class B{
397  public:
398      virtual void f() = 0;
399  };
400
401  class D: public B
402  {
403  public:
404      void f(){cout<<"D\n";}
405  };
406
407  int main()
408  {
409      ///B ob; /// nu se pot instantia clasele abstracte
410      /// B *ob; /// pointeri da
411      /// D ob; /// daca f nu e definita in D, atunci si D devine abstracta
412      D ob;
413      ob.f();
414  }

```

OBS - excepție! Destructorii virtuali puri - trebuie redefiniți în afara clasei de bază - deci nu mai este nevoie să îl definesc în clasa derivată.

```

419  /// Exemplificare clasa abstracta
420  class B
421  {
422  public:
423      virtual void f() = 0;
424  };
425
426  void B::f() redefinesc f-ul
427  {
428      cout<<"B\n";
429  }
430
431  class D: public B
432  {
433  public:
434      void f()
435      {
436          B::f(); folosesc funcția f care a fost redefinită
437          cout<<"D\n";
438      }
439  };
440
441  int main()
442  {
443      D ob; pot să instanțiez D-ul
444  }

```

Programul rulează fără niciun fel de problemă

Atenție!!! Chiar dacă am redefinit f-ul în afara clasei B, clasa B rămâne abstractă și nu pot să o instanțiez!!!



```

419  /// Exemplificare clasa abstracta
420  class B
421  {
422  public:
423      virtual void f() = 0;  asta este ceea ce se transmite
424  };
425
426  void B::f()
427  {
428      cout<<"f\n";
429      asta e local
430  }
431
432  class D: public B
433  {
434  public:
435      /*void f()
436      {
437          B::f();
438          cout<<"D\n";
439      }*/
440  };
441
442  int main()
443  {
444      D ob;
445      ob.f();
446  }

```

tot clasă abstractă rămâne!

Corpul este într-o zonă de memorie... dar totnull este adresa funcției în vtable

vtable-ul pt. clasa derivată conține tot null pe prima poziție la funcția f pt. că nu am redefinit funcția

logs & others

Code::Blocks Search results Build log Build messages x Debugger

File	Line	Message
C:\Users\Ank\D...		In function 'int main()':
C:\Users\Ank\D...	443	error: cannot declare variable 'ob' to be of abstract type 'D'

Atenție!!! Tabelul vtable se modifică doar dacă modific eu funcția, altfel se moștenește din clasa de bază.

## 2.3 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

## 2.4 Constructorii si virtualizare

Obs. NU putem avea constructorii virtuali!!!

(Eu ca să fac un obiect derivat, întâi trebuie să existe constructorul clasei de bază)

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.

## 2.5 Destructori si virtualizare

```

1  class Base1 {public: ~Base1() { cout << "~Base1()\n"; } };
2
3  class Derived1 : public Base1 {public: ~Derived1() { cout << "~Derived1()\n"; } };
4
5  class Base2 {public:
6      virtual ~Base2() { cout << "~Base2()\n"; }
7  };

```



```

8
9 class Derived2 : public Base2 {public: ~Derived2() { cout << "~Derived2()\n"; } };
10
11 int main() {
12     Base1* bp = new Derived1;
13     delete bp; // Afis: ~Base1()
14     Base2* b2p = new Derived2;
15     delete b2p; // Afis: ~Derived2() ~Base2()
16 }

```

Listing 1: Exemplu Slide 59

## 2.6 Destructori virtuali puri

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

Restricție: trebuiesc definiți în clasă (chiar dacă este abstractă).

La moștenire nu mai trebuiesc 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.

```

1 class AbstractBase {
2 public:
3     virtual ~AbstractBase() = 0;
4 };
5
6 AbstractBase::~AbstractBase() {}
7
8 class Derived : public AbstractBase {};
9 // No overriding of destructor necessary?
10 int main() { Derived d; }

```

Listing 2: Exemplu

## 2.7 Funcții virtuale în 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.

```

1 class Base { public:
2     virtual ~Base() { cout << "Base1()\n"; f(); }
3     virtual void f() { cout << "Base::f()\n"; }
4 };
5 class Derived : public Base { public:
6     ~Derived() { cout << "~Derived()\n"; }
7     void f() { cout << "Derived::f()\n"; }
8 };
9
10 int main() {
11     Base* bp = new Derived; // Afis: ~Derived() Base1() Base::f()
12     delete bp;
13 }

```

Listing 3: Exemplu