POO. Curs 7

Anca Dobrovăț și Andrei Păun. Document editat de Raluca Tudor

1 Aprilie 2020

Obs. A se vedea capitolele 14 (Inheritance & Composition) și 15 (Polymorphism & Virtual Functions) din B. Eckel, Thinking in C++.

Câteva observații: numele funcției este, de fapt, un pointer constant către o zonă de memorie. Dacă "const" este mai apropiat de tipul returnat de funcție, atunci pointerul reține ceva constant. Dacă nu, atunci pointerul este constant! Depinde cum se raportează "const" față de numele funcției! (vezi cursurile pt funcții constante)

1 Proiectarea descendentă a claselor. Moștenirea în C++

1.1 Ordinea chemării constructorilor și destructorilor

La fiecare nivel se apelează:

1. întâi constructorul de la moştenire (constructorul clasei de bază - nu pot avea un obiect derivat dacă nu am deja creat un obiect de bază!),
2. apoi constructorii din obiectele membru (constructorii subobiectelor) în clasa respectivă, care sunt chemați în ordinea definirii,
3. și la final, constructorul propriu.

Destructorii sunt chemați în ordinea inversă a constructorilor

```
class cls {
    int x;
public:
    cls(int i=0) {
        cout << "Inside constructor 1" << endl;
        x=i;
    }
    ~cls() {
        cout << "Inside destructor 1" << endl;
    }
};

class clss {
    int x;
    cls xx;
public:
    clss(int i=0) {
        cout << "Inside constructor 2" << endl;
        x=i;
    }
    ~clss() {
        cout << "Inside destructor 2" << endl;
    }
};

class clss2 {
    int x;
    clss xx;
    cls xxx;
public:
```

```
    clss2(int i=0) {
        cout << "Inside constructor 3" << endl;
        x=i;
    }
    ~clss2() {
        cout << "Inside destructor 3" << endl;
    }
};

int main() {
    clss2 s;
}
```

Listing 1: Exemplu Slide 16 (examen)

OUTPUT:
Inside constructor 1
Inside constructor 2
Inside constructor 1
Inside constructor 3
Inside destructor 3
Inside destructor 1
Inside destructor 2
Inside destructor 1

```
#define CLASS(ID) class ID { \  // macrodefinitie
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;
}
```

Listing 2: Exemplu Slide 18

OUTPUT:
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

1.2 Redefinirea funcţiilor membre

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

Este permisă supradefinirea funcţiilor membre clasei de bază cu funcţii membre ale clasei derivate. 2 modalităţi de a redefini o funcţie membră:

- **cu acelasi antet ca în clasa de bază ("redefining" - în cazul funcţiilor oarecare / "overloading" - în cazul funcţiilor virtuale);**
- **cu schimbarea listei de argumente sau a 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"
}
```

Listing 3: Exemplu Slide 21 - pastrarea 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); }
```

Listing 4: Exemplu Slide 22 - nepastrarea antetului/tipului returnat

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

```
class Base {
public:
    int f() const {
        cout << "Base::f()\n";  return 1;
    }
    int f(string) const {  // observati ca am deja supraincarcare in interiorul
    clasei de baza
        return 1;
    }
    void g() { }
};

class Derived1 : public Base {
public:
    void g() const {} //rescriu functia g, dar fara sa modific lista de parametri/
    tipul returnat
    // f se va pastra ca in clasa de baza
};

class Derived2 : public Base {
public:
    // Redefinition:
    int f() const {
        cout << "Derived2::f()\n";
        return 2;
    }
    // nu redefinesc f(string)
    // dar pt ca am redefinit f() => toate f-urile vor fi ascunse!
};

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
}
```

Listing 5: Exemplu Slide 23 - B. Eckel & Examen

Obs:

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

Scopul principal al moştenirii: polimorfismul.

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

Obs.2:

Ce este o interfață? Interfața, pe de o parte, se referă la funcțiile care sunt implementate într-o clasă (în speță, slideul 25 - funcțiile membre din clasa de bază, deci nu ceea ce conține obiectul, ci ceea ce poate face cu ele). Pe de altă parte, în contextul claselor abstracte, interfața se referă la o întreagă clasă de bază din care nu pot să creez obiecte, dar pot să derivez din ea.

Particularități la funcții

- **constructorii și destructorii nu sunt moșteniți** (se redefiniesc noi constr. și destr. pentru clasa derivată)

- similar **operatorul DE ATRIBUIRE =** (un fel de constructor) - *vezi exemplul cu forma*

```
class Forma {
protected:
    int h;
public:
    Forma& operator=(const Forma& ob) {
        if (this!=&ob) {
            h = ob.h;
        }
        return *this;
    }
};

class Cerc: public Forma  {
protected:
    float raza;
public:
    Cerc& operator=(const Cerc& ob) {
        if (this != &ob) {
            this->Forma::operator=(ob);
        }
        return *this;
    }
};
```

Listing 6: Exemplu cu Forma

1.3 Moştenirea si funcţiile 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 signaturii unei funcții din clasa de bază duce la ascunderea celorlaltor versiuni ale funcției.

Dar: O funcție membră statică nu poate fi virtuală!!!

```
class Base {
public:
    static void f()     {        cout << "Base::f()\n";    }
    static void g()    {        cout << "Base::g()\n";    }
  };

class Derived : public Base {
public:    // Change argument list:
    static void f(int x)    {        cout << "Derived::f(x)\n";    }
};

int main() {
    int x;
    Derived::f(); // ascunsa de supradefinirea f(x)
    Derived::f(x);
    Derived::g();
}
```

Listing 7: Exemplu Slide 29

2 Moştenirea cu specificatorul “private”

- inclusă în limbaj pentru completitudine;

- este mai bine a se utiliza compunerea în locul moştenirii 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 moştenire).
- dacă în clasa de bază o componentă era public, iar moştenirea se face cu specificatorul private, **se poate reveni la public utilizând:**

using Baza::nume_componenta

```
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
    // - ambele functii supraincarcate redevin publice in clasa derivata
};

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

Listing 8: Exemplu Slide 33 - B. Eckel

2.1 Moştenire multiplă (MM)

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

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!!

Moştenire multiplă: ambiguități (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???  // SPUNEM EXPLICIT ob.derived1::i !!!
ob.j = 20;
ob.k = 30;
ob.sum = ob.i + ob.j + ob.k; // i ambiguous here, too
```

```
12 cout << ob.i << " "; // also ambiguous, which i?
13 cout << ob.j << " " << ob.k << " ";
14 cout << ob.sum;
15 return 0;
16 }
```

Listing 9: Exemplu Slide 40

Dar dacă avem nevoie doar de o copie lui i?
Nu vrem să consumăm spaţiu în memorie.
Folosim MOȘTENIRE VIRTUALĂ:

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

Dacă avem moştenire de două sau mai multe ori dintr-o clasă de bază (**fiecare moştenire trebuie să fie virtuală**) atunci compilatorul alocă spaţiu pentru o singură copie;

În clasele derived1 şi 2 moştenirea e la fel ca mai înainte (niciun efect pentru virtual în acel caz)

Atenție! Acum al cărei clase este i-ul moștenit de clasa derivată 3? Răspuns: al clasei derivate 1! Pot să fac așa încât i-ul să fie al clasei derivate 2? Da, schimb ordinea - adică class derived3: public derived2, public derived2 {...}

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

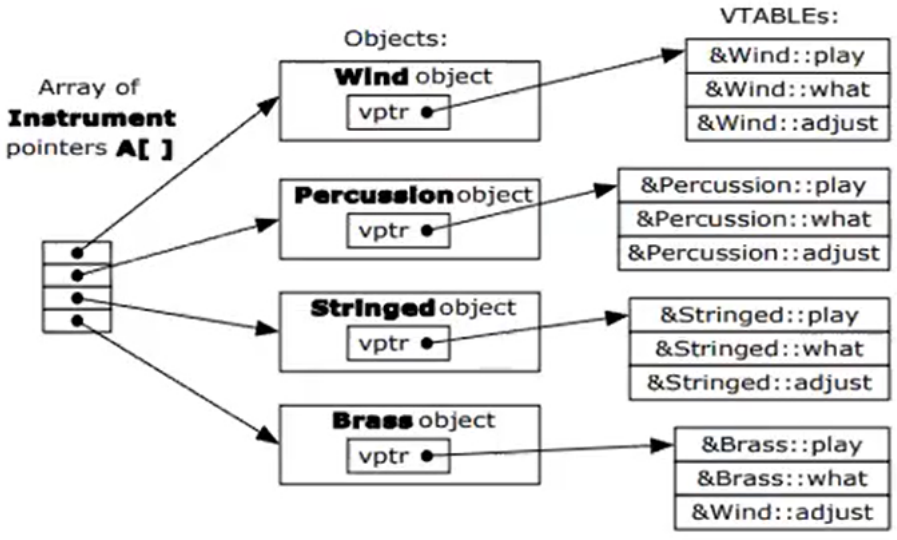
3.1 Parametrizarea metodelor (polimorfism la execuție).

3.2 Funcții virtuale în C++.

La funcțiile virtuale, keyword-ul virtual - ne dă posibilitatea de a stabili cu ce tip de date lucrăm în timpul execuției (tipul de date nu este stabilit la compilare) - s.n. POLIMORFISM LA EXECUȚIE.

În plus, prin keyword-ul virtual, se mai adaugă încă 1 POINTER la clasa mea - către adresele funcțiilor respective!

ZONA DE MEMORIE UNDE SUNT CREATE FUNCȚIILE NU APARȚINE CLASEI. Însă, în momentul în care folosesc funcții virtuale, 1 POINTER aparține de clasă către zonele respective de memorie!



(Vezi și OOP Observații - obs. 1.)

Obs. Faptul că definesc dacă o funcție este virtuală sau nevirtuală nu mă afectează dacă eu folosesc tipuri de date statice. Dacă folosesc pointeri, atunci văd utilitatea funcțiilor virtuale!

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

Funcţii virtuale: ne lasă să chemăm funcţiile pentru tipul derivat.

```
#include <iostream>

using namespace std;

class Baza
{
    public:
    virtual void afis(){ cout<<"B";}
};

class Derivata1: public Baza
{
    public:
    void afis(){ cout<<"D1";}
};

class Derivata2: public Baza
{
    public:
   // void afis(){ cout<<"D2";}
};

int main()
{
    Baza *p;
    Derivata1 ob1;
    Derivata2 ob2;

    p = &ob1;
    p->afis();  // D1 (fara virtual se va afisa B)
    p = &ob2;
    p->afis();  // B

}
```

Listing 10: Exemplu Curs Anca Dobrovăț

Cum funcționează? vptr-ul se copiază și în clasa derivată. În momentul în care s-a copiat în clasa derivată, am un alt pointer vptr care face legătura către funcția (în speță) afiș din clasa de bază. Dacă funcția afiș este rescrisă, atunci vptr va reține adresa noii funcții.

(Numele funcției este un pointer către zona de memorie unde se află funcția -> pointer către acel vtable unde se află vectorul de adrese ale funcțiilor 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 function:
  void play(note) const {
    cout << "Wind::play" << endl;  }
};
void tune(Instrument& i) {  i.play(middleC);  }
```

```
15
16 int main() {
17   Wind flute;
18   tune(flute); // se afiseaza Wind::play
19 }
```

Listing 11: Exemplu Slide 46

LATE BINDING
Faptul că folosesc keyword-ul virtual în funcții îmi spune cum să se comporte codul - se face LATE BINDING - nu decid tipurile de date reținute în anumiți pointeri de la început (la compilare) - îi dau libertatea să îmi spună ce tip de date folosesc abia LA EXECUȚIE.

3.3 Clase abstracte.

3.4 Overloading pe funcții virtuale.

3.5 Destructori și virtualizare.