

Tutoriat 4 POO

Bianca-Mihaela Stan, Silviu Stăncioiu

April 2021

STOP DOING OOP

- STRUCTURES WERE NOT SUPPOSED TO BE GIVEN FUNCTIONS
- YEARS OF CLASSES yet NO REAL-WORLD USE FOUND for using friend classes
- Wanted to access private variables from other objects anyway for a laugh? We had a tool for that: It was called “PUBLIC”
- “Yes please give me abstract class. Please give me diamond inheritance” - Statements dreamed up by the utterly Deranged

LOOK at what object oriented programmers have been demanding your Respect for all this time, with all the compilers & editors we built for them
(This is REAL OOP , done by REAL Object oriented programmers):

```
class A
{
    int i;
protected: static int x;
public: A(int j=7) {i=j;x=j;}
int get_x() {return x;}
int set_x(int j) {int y=x; x=j; return y;}
A operator=(A a1) {set_x(a1.get_x()); return a1;}
};

int A::x=15;
```

```
#include<iostream>
using namespace std;
class A
{
    int valoare;
public: A(int param=3):valoare(param){}
    int getValoare(){return this->valoare;}
};
int main()
{
    A vector[]={{new A(3)},*(new A(4)),*(new A(5)),*(new A(6))};
    cout<<vector[2].getValoare();
    return 0;
}
```

```
class A
{
    int i;
public: A() {i=1;}
    virtual int get_i() {return i;}
};
class B: public A
{
    int j;
public: B() {j=2;}
    int get_i() { return A::get_i()+j;}
};
```

?????

???????

?????????????????????

“Hello I would like template<typename T> apples please”
They have played us for absolute fools

1 Moștenirea multiplă

În C++, putem face ca o clasă să moștenească mai multe clase de bază. Acest lucru se numește moștenire multiplă, și este exact ce ne așteptăm să fie, clasa derivată primește tot ce se află în clasele de bază, după aceleași reguli ca la moștenirea simplă. Exemplu:

```
#include <iostream>

using namespace std;

class base1 // una din clasele de baza
{
public:
    base1()
    {
        cout << "Constructor base1" << endl; // afiseaza un mesaj cand se
                                                // apeleaza constructorul clasei de
                                                // baza
    }
};

class base2 // a doua clasa de baza
{
public:
    base2()
    {
        cout << "Constructor base2" << endl; // afisam un mesaj cand se apeleaza
                                                // constructorul acestei clase
    }
};

class derived : public base1, public base2 // clasa noastră nouă moșteneste
                                            // atât clasa base1, cât și clasa
                                            // base2. ordinea în care se trec
                                            // clasele din care se moștenesc
                                            // contează. În funcție de aceasta
                                            // ordine se decide ordinea în care
                                            // se apelează constructorii clasei
                                            // de bază atunci cand se
                                            // construiește un obiect din
                                            // aceasta clasa derivată.
{
public:
    derived() :
        base2(), // apelam constructorii claselor de
                  // baza. În acest caz nu este nevoie
                  // să ii apelam constructorii,
                  // deoarece acestia nu au argumente,
                  // deci oricum se apelau. Se observă
                  // că în lista de initializare
                  // prima oară am zis că apelam
                  // constructorul lui base2, apoi
                  // constructorul lui base1. Aceasta
                  // ordine nu are nicio importanță,
                  // constructorii claselor de baza se
        base1()
}
```

```

        // apeleaza in functie de ordinea
        // in care sunt trecuti sus, in
        // declararea clasei. Se observa
        // ca pe ecran prima oara se
        // afiseaza ca se apeleaza
        // constructorul lui base1, iar apoi
        // constructorul lui base2

    {
        cout << "Constructor derived" << endl; // in constructorul obiectului
                                                // derivat afisam un mesaj.
    }
};

int main()
{
    derived d;                                // va afisa pe ecran:
                                                // Constructor base1
                                                // Constructor base2
                                                // Constructor derived

    return 0;
}

```

Observație:

Destructorii sunt apelați în ordinea inversă a constructorilor.

Acum, este posibil să avem cazuri în care avem variabile sau metode cu același nume în ambele clase deriveate, iar noi ne dorim să le folosim în interiorul clasei noastre deriveate. Fie următorul exemplu:

```

#include <iostream>

using namespace std;

class base1           // clasa base1 are o variabila membra numita x
{
public:
    int x;           // variabila membra x

    base1()
    {
        x = 5;       // in constructorul clasei ii atribuim valoarea 5 lui x
    }
};

class base2           // clasa base2, care are si ea o variabila membra
                      // numita x
{
public:
    int x;           // variabila membra x

    base2()
    {
        x = 10;      // in constructorul clasei base2 ii atribuim valoarea
                      // 10 lui x
    }
};

```

```

    }

};

class derived : public base1, public base2
{
public:
    derived()
    {
        cout << x << endl; // vrem sa il afisam pe x pe ecran. ce se va afisa?
        // 5 sau 10? Well, in clasa noastră derivată acum există
        // și x-ul din clasa base1, dar și cel din clasa base2.
        // La aceasta linie avem eroare de compilare, deoarece
        // compilerul nu știe dacă ne referim la x-ul din clasa
        // base1, sau cel din clasa base2.
    }
};

int main()
{
    derived d;

    return 0;
}

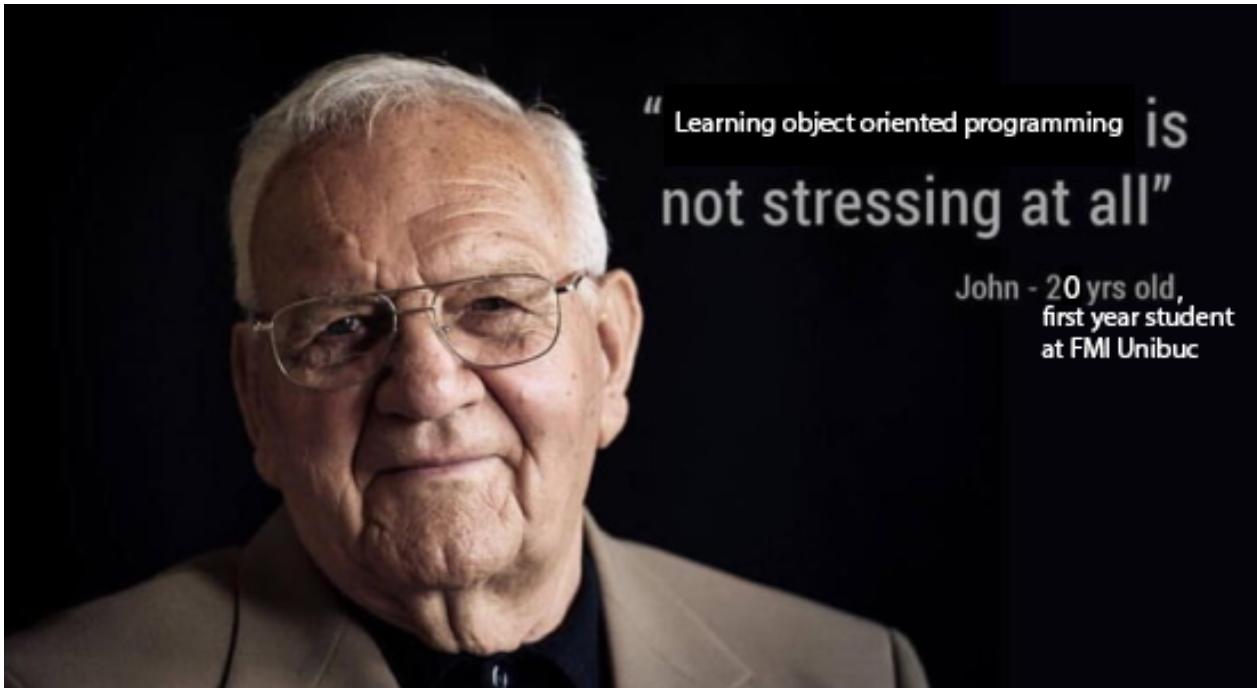
```

În exemplul de mai sus, avem o eroare de compilare, deoarece compilerul nu știe la ce x ne referim atunci când îi spunem că vrem să-l afișăm pe ecran în clasa derivată. Pentru a repara această problemă trebuie să îi spunem la care x ne referim. Pentru a face asta, putem înlocui acel cout cu:

```

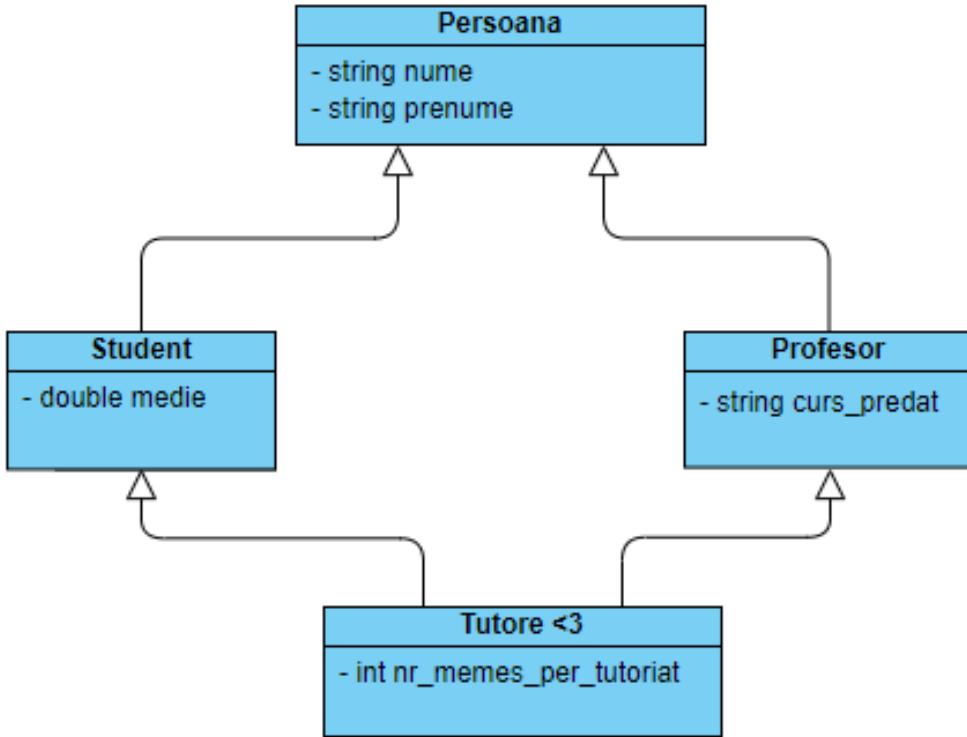
cout << base1::x << endl; // in caz ca dorim sa afisam x-ul din base1
                           // SAAAAAAAAAU
cout << base2::x << endl; // in caz ca dorim sa afisam x-ul din base2

```



2 Moștenire în diamant

Dacă avem o ierarhie de genul asta:



Codul corespunzător este:

```
#include <iostream>
#include <cstring>

using namespace std;

class Persoana
{
protected:
    string _nume;
    string _prenume;
public:
    Persoana(string nume = "", string prenume = "") :
        _nume(nume), _prenume(prenume) {}
};

class Student : public Persoana
{
protected:
    double _medie;
public:
    Student(double medie = 0, string nume = "", string prenume = "") :
        Persoana(nume, prenume), _medie(medie) {}
};

class Profesor : public Persoana
{
protected:
    string curs_predat;
public:
    Profesor(string curs_predat) :
        Persoana(curs_predat) {}
};

class Tutore <3
{
protected:
    int nr_memes_per_tutoriat;
public:
    Tutore <3(int nr_memes_per_tutoriat) :
        nr_memes_per_tutoriat(nr_memes_per_tutoriat) {}
};
```

```

};

class Profesor : public Persoana
{
protected:
    string _curs_predat;
public:
    Profesor(string curs_predat="", string nume="", string prenume="") :
        Persoana(nume, prenume), _curs_predat(curs_predat) {}
};

class Turore : public Student, public Profesor
{
private:
    int _nr_memes_per_tutoriat;
public:
    Turore(int nr_memes_per_tutoriat=0, double medie = 0,
           string curs_predat="", string nume="", string prenume="")
        : Student(medie, nume, prenume),
          Profesor(curs_predat, nume, prenume),
          _nr_memes_per_tutoriat(nr_memes_per_tutoriat) {}

    void set_nume(string nume)
    {
        _nume = nume; // eroare: "Turore::_nume" is ambiguous ()
    }
};

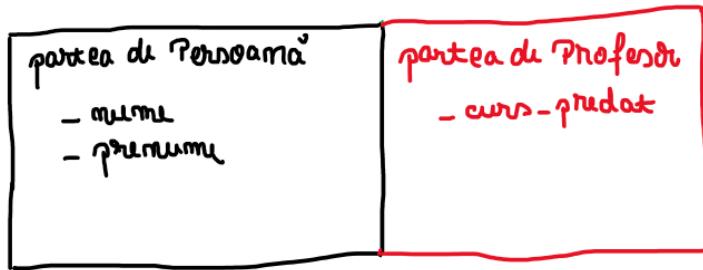
int main()
{
    Student student;
    Profesor profesor;
    Turore tutore;
    return 0;
}

```

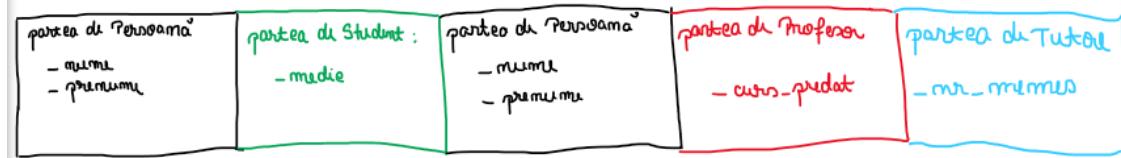
Si vedem si prima problema, `Turore::_nume` is ambiguous. De ce se intampla asta? Sa ne amintim cum arata un obiect din clasa Student:



Si cum arata un obiect de tip Profesor:



Să construim acum obiectul de tip `Tutore`. El are partea de `Student` + partea de `Profesor` + partea de `Tutore`, adică:



Do we see the problem here? Un obiect de tip `Tutore` are 2 parti de tip `Persoana`. Atunci cand apelam `_nume` programul nu stie pe care parte sa o aleaga. Cum rezolvam aceasta problema? Mostenirea virtuala.

3 Mostenirea virtuala

Mostenirea virtuala ne asigura ca obiectul care mostenește virtual dintr-o clasa va avea o singură parte de baza (nu vom avea situația de mai sus cu 2 parti de baza).

```
#include <iostream>
#include <cstring>

using namespace std;

class Persoana
{
protected:
    string _nume;
    string _prenume;
public:
    Persoana(string nume = "", string prenume = "") :
        _nume(nume), _prenume(prenume) {}
};

class Student : virtual public Persoana           // mostenire virtuala
{
protected:
    double _medie;
public:
    Student(double medie = 0, string nume = "", string prenume = "") :
        Persoana(nume, prenume),               // linia asta trebuie scrisa
                                                // ca sa putem crea obiecte
                                                // de tip Student, dar este
                                                // ignorata cand cream obiecte
                                                // de tip Tutore
        _medie(medie) {}
};
```

```

class Profesor : virtual public Persoana           // mostenire virtuala
{
protected:
    string _curs_predat;
public:
    Profesor(string curs_predat="", string nume="", string prenume="") :
        Persoana(nume, prenume),                         // la fel ca la constructorul
                                                       // din Student
        _curs_predat(curs_predat) {}
};

class Tutoare : public Student, public Profesor
{
private:
    int _nr_memes_per_tutoriat;
public:
    Tutoare(int nr_memes_per_tutoriat=0, double medie = 0, string curs_predat="", string nume="", string
        : Persoana(nume, prenume),                         // Aici obiectul de tip Tutoare
                                                       // este responsabil sa isi apeleze
                                                       // singur baza de tip Persoana.
                                                       // Aceasta este o situatie in care
                                                       // obiectul are voie sa apeleze
                                                       // constructori care nu sunt parinti
                                                       // directi.
        Student(medie, nume, prenume),
        Profesor(curs_predat, nume, prenume),
        _nr_memes_per_tutoriat(nr_memes_per_tutoriat) {}

    void set_nume(string nume)
    {
        _nume = nume;                                     // Acum am rezolvat eroarea.
    }
};

int main()
{
    Student student;
    Profesor profesor;
    Tutoare tutoare;
    return 0;
}

```

Cateva detalii despre mostenirea virtuala:

- Bazele virtuale se creeaza inaintea tuturor bazelor nevirtuale.
- Student si Profesor inca au codul care apeleaza constructorul pentru Persoana, insa cand se creeaza un obiect de tip tutoare acel cod este pur si simplu ignorat.
- Toate clasele care mostenesc dintr-o baza virtuala primesc un vtable si un pointer catre acest vtable, chiar daca in conditii normale nu ar fi primit. (aka nu contine metode virtuale)



4 Destructorul virtual

Atunci cand alocam memorie dinamic in clasele noastre avem nevoie sa definim propriul destructor in clasele noastre. Sa luam cazul:

```
#include <iostream>
using namespace std;

class baza
{
private:
    int* _a;
    int _n;
public:
    baza(int n =0)
    {
        cout<<"se aloca memorie dinamic pentru _a\n";
        _a = new int[n];
        _n = n;
    }
    ~baza()
    {
        cout<<"se distruce _a-ul\n";
        delete[] _a;
    }
};
```

```

class derivata : public baza
{
private:
    int* _b;
    int _m;
public:
    derivata(int n=0, int m=0) : baza(n)
    {
        cout<<"se aloca memorie dinamic pentru _b\n";
        _b = new int[m];
        _m = m;
    }
    ~derivata()
    {
        cout<<"se distrue _b-ul\n";
        delete[] _b;
    }
};

int main()
{
    derivata* derived_object = new derivata(4, 5);
    baza* ptr = derived_object;
    delete ptr;
    return 0;
}

```

I hope we all see the problem here. Se aloca memorie pentru `_b`, dar nu se si dezaloca. De ce se intampla asta? `ptr` este un pointer de tip `baza`. Cand se apeleaza `delete` pe el, se apeleaza destructorul de tipul pointerului, adica cel din `baza`. Compilatorul se uita sa vada daca destructorul din `baza` e virtual. Vede ca nu e, ceea ce inseamna ca nu trebuie sa mai apeleze nimic altceva.

Cum rezolvam problema asta? Destructorul virtual.

```

#include <iostream>
using namespace std;

class vector
{
private:
    int* _a;
    int _n;
public:
    vector(int n =0)
    {
        cout<<"se aloca memorie dinamic pentru _a\n";
        _a = new int[n];
        _n = n;
    }
    virtual ~vector() // Acum cand compilatorul vede
                      // ca destructorul e virtual
                      // se uita in vtable si vede
                      // ce destructor trebuie sa mai
                      // apeleze.
    {
        cout<<"se distrue _a-ul\n";
    }
}

```

```

        delete[] _a;
    }
};

class special : public vector
{
private:
    int* _b;
    int _m;
public:
    special(int n=0, int m=0) : vector(n)
    {
        cout<<"se aloca memorie dinamic pentru _b\n";
        _b = new int[m];
        _m = m;
    }
    ~special()
    {
        cout<<"se distrugе _b-ul\n";
        delete[] _b;
    }
};

int main()
{
    special* special_object = new special(4, 5);
    vector* ptr = special_object;
    delete ptr;                                // Acum se afiseaza:
                                                // se aloca memorie dinamic pentru _a
                                                // se aloca memorie dinamic pentru _b
                                                // se distrugе _b-ul
                                                // se distrugе _a-ul
    return 0;
}

```

Regula: Cand aveti mostenire, folositi intotdeauna destructori virtuali pentru baze.

Profesorii de POO explicând că sunt indulgenți, deși în generațiile anterioare s-a picat pe capete la examen



5 Destructorul pur virtual

Chiar și destructorii pot fi pur virtuali. Reamintim ce inseamnă pur virtual: o funcție virtuală care nu are implementare (este marcată cu `=0`). Aceasta face ca întreaga clasa să fie abstractă (nu se pot instantia obiecte din ea).

De ce ne-am dori un destructor pur virtual? Dacă nu avem implementare pentru el nu putem pur și simplu să îl lasăm pe cel dat de compilator?

Well, avem 2 motive pentru care vrem destructor pur virtuali:

- Ca să nu fie permisă ar trebui să fie adăugată încă o regula în limbaj, ceea ce nu are rost având în vedere că nu daunează cu nimic.
- Cateodată vrem să facem o clasă abstractă, însă nu avem nicio metodă pe care vrem să o facem pur virtuală. Un destructor pur virtual este de ajuns ca să transforme întoată clasa într-o clasa pur virtuală.

Atenție!Pentru destructorul pur virtual trebuie neapărat să scriem și o implementare **în afara clasei**, chiar dacă este pur virtual (care teoretic asta ar trebui să însemne, că nu are implementare).

De ce trebuie să ii scriem o implementare? Pentru că destructorii se apelează în ordinea inversă a construcților. În exemplul:

```

#include <iostream>
using namespace std;

class base
{
public:
    virtual ~base()=0;
};

base::~base()
{
    cout<<"destructorul pentru base\n";
}

class derived : public base
{
public:
    ~derived()
    {
        cout<<"destructorul pentru derived\n";
    }
};

int main()
{
    derived d;
    return 0;
}

```

se afiseaza:

destructorul pentru derived
destructorul pentru base

Deci destructorul pentru base se apeleaza implicit, daca nu i-am da o implementare nu ar avea ce apela.

De ce trebuie sa scriem implementarea in afara clasei? Pentru ca asa e conventia, avem voie sa scriem o implementare pentru functiile pur virtuale, dar daca o facem trebuie sa o scriem in afara clasei (pentru mai multe detalii vezi Tutoriat 2: Functii pur virtuale cu corp).

6 Exercitii

Boboci la FMI rezolvând subiecte de POO, 2021 color



Exercitiul 1

```
#include <iostream>
using namespace std;

class base
{
public:
    virtual void show()
    {
        cout<<"base";
    }
};

class derived : public base
{
public:
    void show()
    {
        cout<<"derived";
    }
};
```

```

void print(base x)
{
    x.show();
}

int main()
{
    derived d;
    print(d);
}

```

Se afiseaza base. Acesta este un exemplu de object slicing. Obiectul d are si o parte de base si o parte de derived, insa cand apelam functia print, se face un cast automat la partea de baza si pierdem accesul la parea derivata. Asadar, chiar daca functia noastra show era virtuala, nu se afiseaza derived.

Exercitiul 2

```

1 #include <iostream>
2 using namespace std;
3
4 class base
5 {
6 protected:
7     int _x;
8 public:
9     base(int x=3) : _x(x) {}
10    void show()
11    {
12        cout<<"base ";
13        cout<<_x<<"\n";
14    }
15};
16
17 class derived : public base
18 {
19 private:
20     int _y;
21 public:
22     derived(int x, int y=4): base(x),_y(y) {}
23     void show()
24     {
25         cout<<"derived ";
26         cout<<_x<<" "<<_y<<"\n";
27     }
28};
29
30 int main()
31 {
32     derived d1(5, 7);
33     derived d2(9, 1);
34     base& b = d2;
35
36     b=d1;
37     b.show();
38     d1.show();

```

```

39     d2.show();
40     return 0;
41 }

```

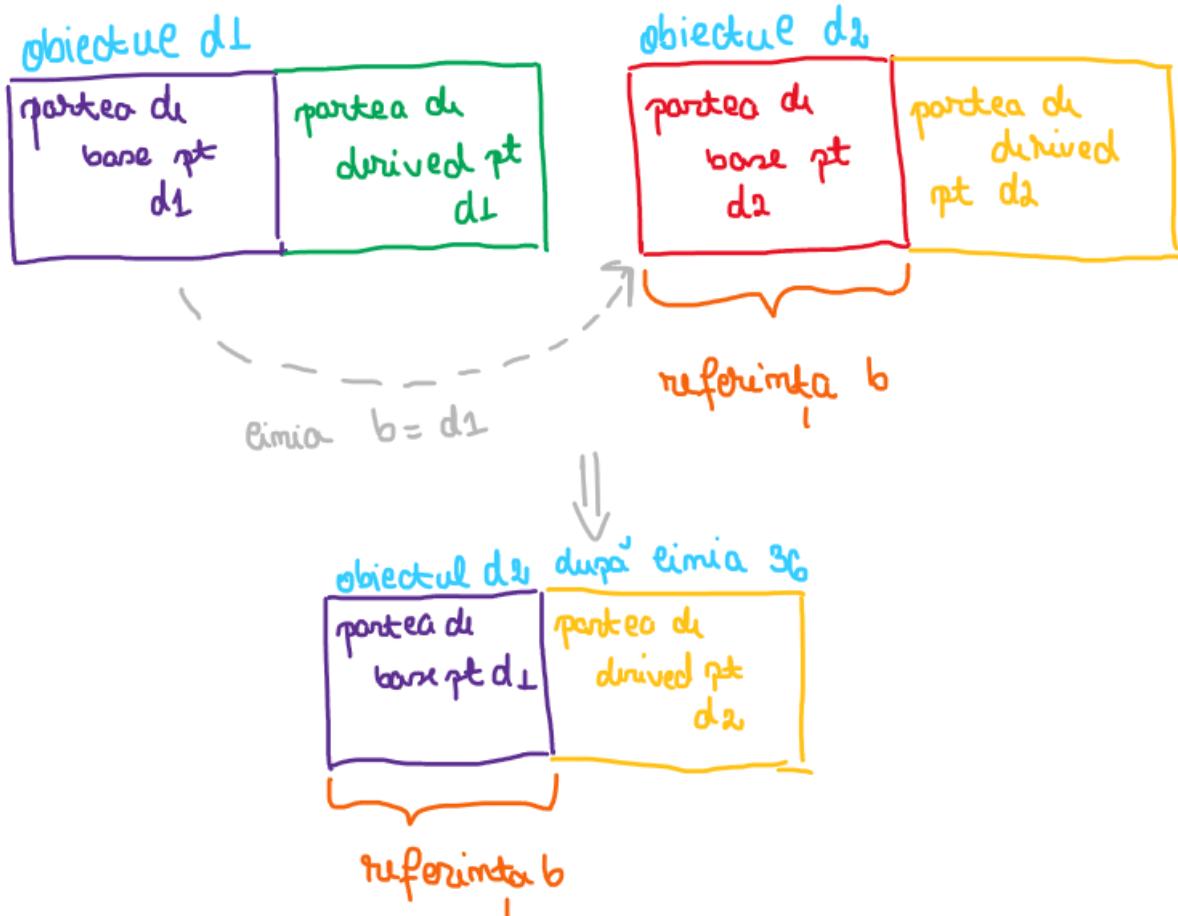
Se afiseaza:

base 5

derived 5 7

derived 5 1

Primele 3 linii se comporta cum ne-am asteptat. Se creeaza 2 obiecte de tip derived, iar b este referinta catre d2, adica alt nume pentru d2. La linia 36 insa, se face copierea doar partii de baza din d1 in partea de baza din d2. De ce? Pentru ca b este de tip base. Se apeleaza operatorul = pentru tipul base(cel implicit dat de compilator), iar acesta nu este virtual. Deci se face copierea doar pe partea de baza a obiectului. Adica:



Acesta este un alt caz de object slicing, am facut un franken-obiect.

Exercitiul 3

```

1 #include <iostream>
2 using namespace std;
3 class A
4 {
5     int i;
6 public:
7     A() {cout<<"A()\n";};
8     A(int k) {cout<<"A(int k)\n"; i = k; }

```

```

9   };
10
11 class B : virtual public A
12 {
13 public:
14     B(){cout<<"B()\n";}
15     B(int i) : A(i) {cout<<"B(int i)\n";}
16 };
17
18 class C : public B
19 {
20 public:
21     C() {cout<<"C()\n";}
22     C(int i) : B(i) {cout<<"C(int i)\n";}
23 };
24
25 int main()
26 {
27     C c(2);
28     return 0;
29 }

```

Se afiseaza:

A()
B(int i)
C(int i)

De ce se afiseaza asa si de ce nu ar fi, cum poate ne-am fi asteptat:

A(int k)
B(int i)
C(int i)

Pentru ca, asa cum am mentionat la mostenirea virtuala, C-ul e responsabil sa isi apeleze constructorul pentru baza virtuala, codul care il apeleaza la linia 15 este ignorat atunci cand construim un obiect de tip C.

Insa la linia 22 C-ul nu apeleaza constructorul pentru baza. Asa ca compilatorul adauga el apelarea constructorului bazei, fara parametri. Deci, pentru compilator, linia 22 este echivalenta cu:

C(int i) : A(), B(i) {cout<<"C(int i)";}

Cum facem sa ne afiseze asta?

A(int k)
B(int i)
C(int i)

Modificam linia 22 astfel: C(int i) : A(i), B(i) {cout<<"C(int i)";}

Exercitiul 4

```

1 #include <iostream>
2 using namespace std;
3 class B
4 {
5 public:
6     B()
7     {
8         cout<<"B ";
9     }

```

```

10     ~B()
11     {
12         cout<<"~B ";
13     }
14 };
15 class D1 : virtual B
16 {
17 public:
18     D1() : B()
19     {
20         cout<<"D1 ";
21     }
22     ~D1()
23     {
24         cout<<"~D1 ";
25     }
26 };
27 class D2 : virtual B
28 {
29 public:
30     D2() : B()
31     {
32         cout<<"D2 ";
33     }
34     ~D2()
35     {
36         cout<<"~D2 ";
37     }
38 };
39 class D3 : B
40 {
41 public:
42     D3() : B()
43     {
44         cout<<"D3 ";
45     }
46     ~D3()
47     {
48         cout<<"~D3 ";
49     }
50 };
51 class D4 : private B
52 {
53 public:
54     D4() : B()
55     {
56         cout<<"D4 ";
57     }
58     ~D4()
59     {
60         cout<<"~D4 ";
61     }
62 };
63 class D5 : virtual public B

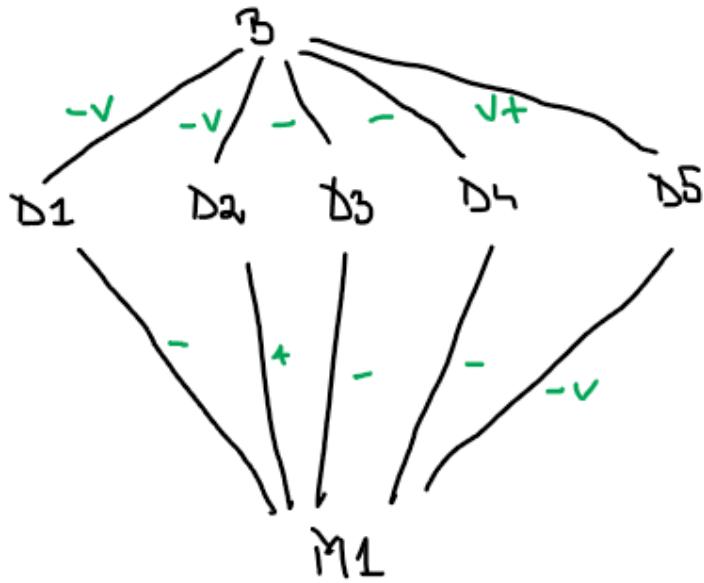
```

```

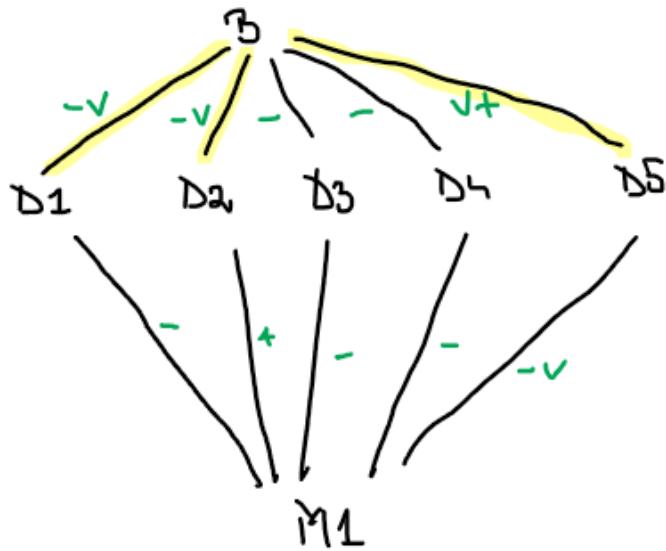
64  {
65  public:
66    D5() : B()
67    {
68      cout<<"D5 ";
69    }
70    ~D5()
71    {
72      cout<<"~D5 ";
73    }
74  };
75  class M1 : D1, public D2, D3, private D4, virtual D5
76  {
77  public:
78    ~M1()
79    {
80      cout<<"\n ~M1";
81    }
82  };
83  class M2 : D1, D2, virtual D3, virtual D4, virtual D5
84  {
85  public:
86    ~M2()
87    {
88      cout<<"\n ~M2";
89    }
90  };
91
92  int main()
93  {
94    M1 m1;
95    cout<<"\n";
96    M2 m2;
97    cout<<"\n";
98    return 0;
99  }

```

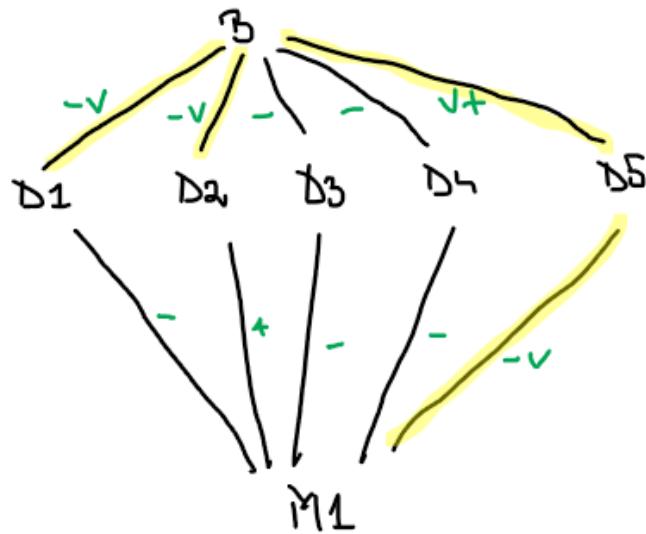
La linia 94 se afiseaza: B D5 D1 D2 B D3 B D4. De ce? Asa arata diagrama pentru M1:



M1 apeleaza constructorul bazei virtuale B si rezolva cele 3 ramuri ale mostenirii marcate aici:



Apoi continu sa isi construiasca bazele virtuale. Cine mai e baza virtuala in afara de B? D5. Apeleaza constructorul lui si s-a mai rezolvat o mostenie:



Mai departe apeleaza constructorii pentru restul bazelor, in ordinea in care se gasesc in mostenire (adica la linia 75). Bazele nerezolvate sunt, in ordinea asta: D1, D2, D3 si D4.

Se apeleaza constructorul pentru D1. Chiar daca D1 apeleaza constructorul pentru baza, aceasta apelare este ignorata (la linia 18) pentru ca construim un obiect de tip M1. Deci se afiseaza doar D1.

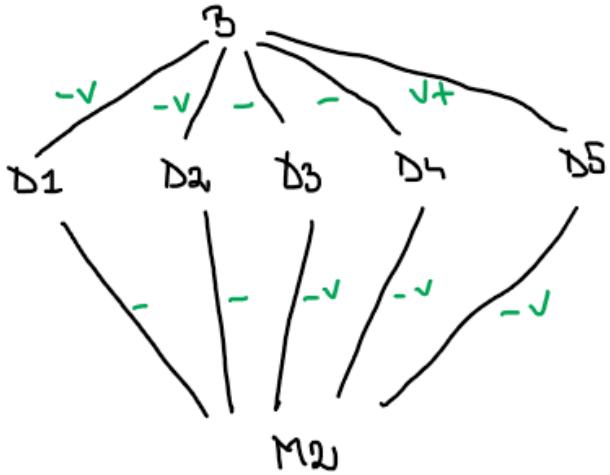
Se apeleaza cosntructorul pentru D2. La fel ca la D1, se ignora apelarea constructorul lui B la linia 30. Deci se afiseaza D2.

Se apeleaza constructorul lui D3. D3 nu mosteneste virtual din baza, deci pentru el suntem obligati sa apelam constructorul bazei. Deci se afiseaza B D3.

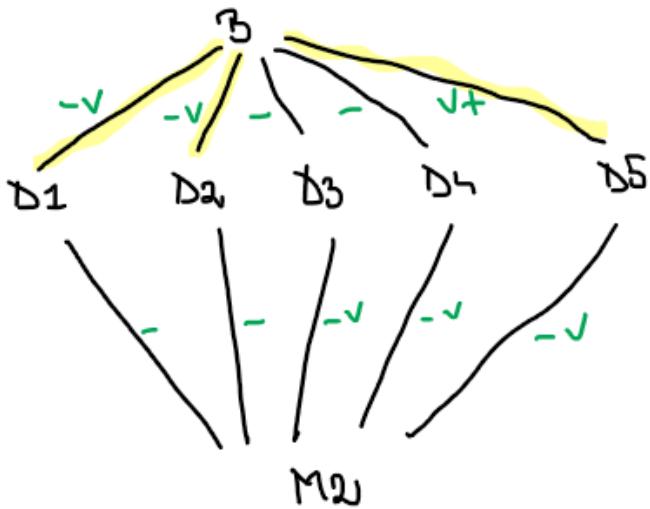
Se apeleaza constructorul lui D4. La fel ca D3, el nu mosteneste virtual din B, deci suntem obligati sa apelam B. Deci se afiseaza B D4.

And that's it, am marcat toate mostenirile.

La linia 96 se afiseaza: B B D3 B D4 D5 D1 D2. De ce? Asa arata arborele pentru M2:



Am zis ca prima data isi rezolva bazele virtuale. Cine sunt baze virtuale? B, D3, D4 si D5. Prima data trebuie sa o rezolve pe B, pentru ca de ea depend restul. Deci se apeleaza constructorul pentru B si se rezolva 3 mosteniri:

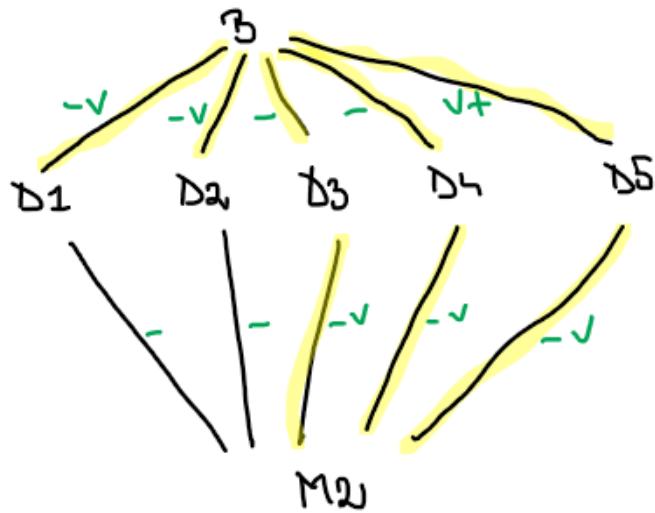


Continuam sa rezolvam bazele virtuale, in ordinea in care apar la mostenirea de la linia 83.

Adica apelam constructorul pentru baza virtuala D3. D3 nu mosteneste virtual din B, deci trebuie sa apelam constructorul bazei din constructorul lui D3. Deci se afiseaza: B D3.

Se apeleaza constructorul pentru urmatoarea baza virtuala, D4. La fel ca la D4, el nu mosteneste virtual din B, deci apelam si B-ul din D4. Deci se afiseaza B D4.

Se apeleaza constructorul pentru urmatoarea baza virtuala, D5. Ea a mostenit virtual din B, deci deja i-am rezolvat baza. Deci se ignora codul care apeleaza constructorul lui B din D5. Deci se afiseaza doar D5. Pana acum mostenirile rezolvate sunt:



Au mai ramas bazele nevirtuale D1 si D2. Se apeleaza constructorul pentru D1 pentru ca el este primul in lista de mostenire de la linia 83. D1 mosteneste virtual din B, deci baza a fost deja rezolvata. Deci se afiseaza doar D1. La fel pentru D2.

Mai departe se afiseaza:

```
M2 D2 D1 D5 D4 B D3 B B
M1 D4 B D3 B D2 D1 D5 B
```

S-au construit, in ordinea asta, obiectele m1 si m2. Cand se distrug se distrug in ordinea inversa fata de cum s-au creat, adica m2, m1.

Cum se distrug? Se apeleaza destructorii lor in ordinea inversa apelarii constructorilor. Deci daca pentru m1 constructorii s-au apelat in ordinea asta:

B B D3 B D4 D5 D1 D2

Destructorii se vor apela in ordinea inversa:

M2 D2 D1 D5 D4 B D3 B B

La fel pentru m1:

B D5 D1 D2 B D3 B D4

M1 D4 B D3 B D2 D1 D5 B

Exercitiul 5

```

1 #include <iostream>
2 using namespace std;
3
4 class A
5 {
6 protected:
7     int x =10;
8 public:
9     virtual void print()
10    {
11        cout<<x<<" ";
12    }

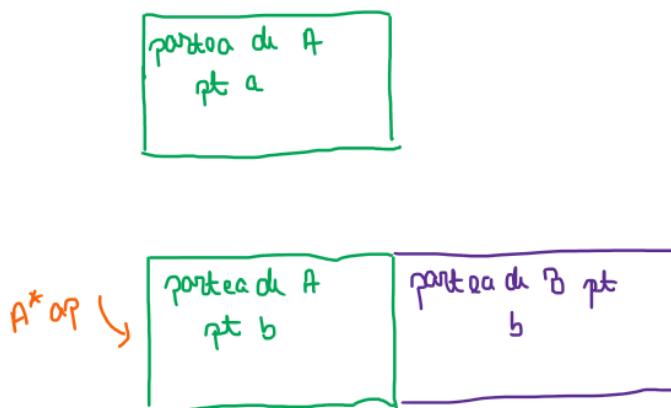
```

```

13  };
14  class B : public A
15  {
16  private:
17      int y = 7;
18  public:
19      void print()
20      {
21          A::print();
22          cout << y << " ";
23      }
24  };
25  class C
26  {
27  private:
28      int z = 3;
29  public:
30      void print()
31      {
32          cout << z << " ";
33      }
34  };
35
36  int main()
37  {
38      A a;
39      B b;
40
41      A* ap = &b;
42      if(ap!=nullptr)
43          ap->print();
44      B* b1 = dynamic_cast<B*> (&a);
45      if(b1!=nullptr)
46          b1->print();
47      B* b2 = dynamic_cast<B*> (ap);
48      if(b2!=nullptr)
49          b2->print();
50      C* c = dynamic_cast<C*> (ap);
51      if(c!=nullptr)
52          c->print();
53
54      A& ar = dynamic_cast<A&> (*ap);
55      ar.print();
56      B& br = dynamic_cast<B&> (*ap);
57      br.print();
58      C& cr = dynamic_cast<C&> (*ap);
59      cr.print();
60      return 0;
61  }

```

Asa arata obiectele noastre:



La linia 43 se afiseaza 10 7 pentru ca functia print e virtuala.

La linia 44 se incercă crearea unui pointer de tip B la un obiect de tip a. Nu se poate face chestia asta, deci dynamic_cast-ul returnează nullptr.

La linia 47 se incercă crearea unui pointer de tip B dintr-un pointer de tip A la un obiect de tip B. Acest cast e valid, deci o sa se returneze un pointer de tip B la obiectul b. Iar la linia 49 se afiseaza 10 7.

La linia 50 se incercă crearea unui pointer de tip C dintr-un pointer de tip A. C si A nu sunt in aceeasi ierarhie, deci cast-ul nu e valid si dynamic_cast-ul va returna nullptr.

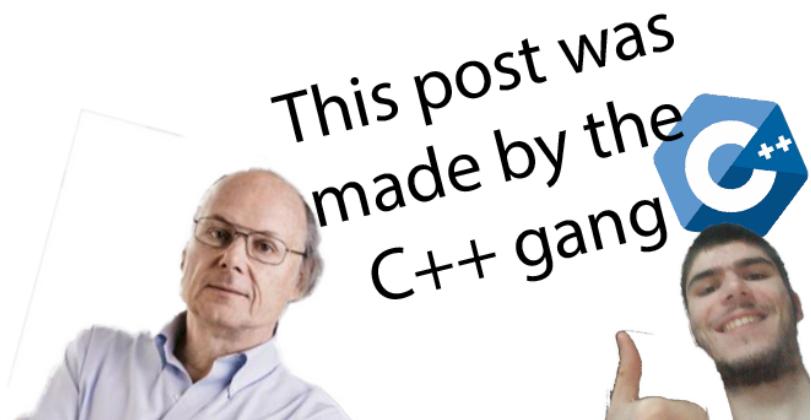
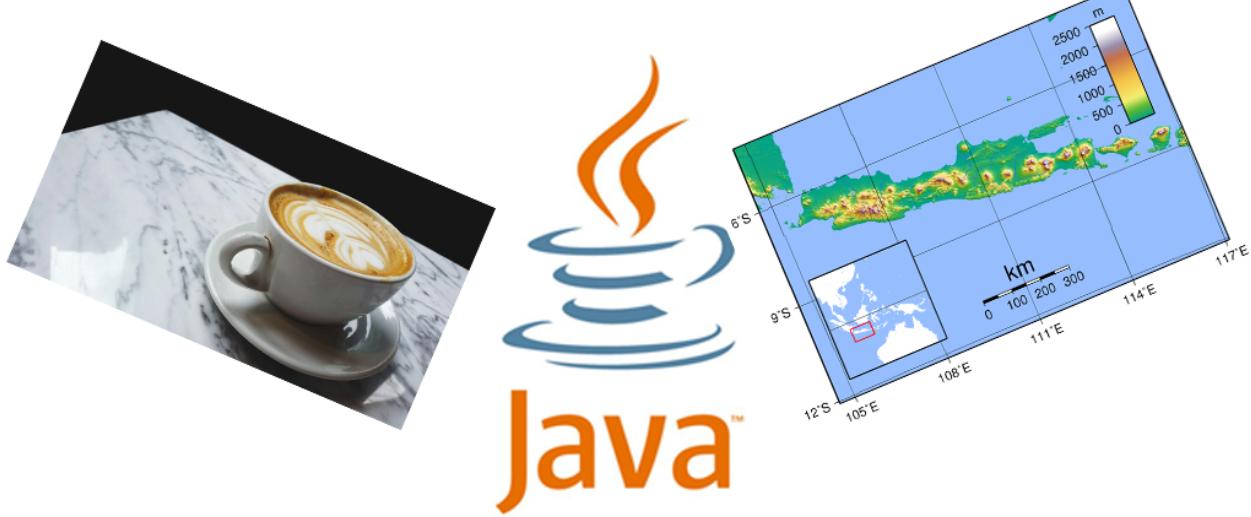
La linia 54 se incercă crearea unei referinte de tip A la un obiect de tip B. Acest acst este valid, iar la linia 55 se afiseaza 10 7 pentru ca print este virtual.

La linia 57 se incercă crearea unei referinte de tip B catre un obiect de tip B, iar la linia 57 se afiseaza 10 7.

La linia 58 se incercă crearea unei referinte de tip C la un obiect de tip B. C si B nu sunt in aceeasi ierarhie deci acest cast nu este valid. Daca aveam pointeri, dynamic_cast ar fi returnat nullptr. Nu putem avea insa null reference. Asa ca s-a introdus in C++ eroare std::bad_cast. Asta primim la linia 58:
terminate called after throwing an instance of 'std::bad_cast'

Ce fel de eroare este asta? La run time. Putem observa aceasta rulare doar dupa ce s-a alocat memoria pentru obiecte si putem afla care este tipul real al obiectelor prin RTTI (vtables si vpointers).

Imagine using Java



Exercitiul 6

```
1 #include <iostream>
2 using namespace std;
3
4 class A
5 {
6 protected:
7     int x =10;
8 public:
9     virtual void print()
10    {
11         cout<<x<<" ";
12    }
```

```

13  };
14  class B : public A
15  {
16  private:
17      int y = 7;
18  public:
19      void print()
20      {
21          A::print();
22          cout << y << " ";
23      }
24  };
25  class C
26  {
27  private:
28      int z = 3;
29  public:
30      void print()
31      {
32          cout << z << " ";
33      }
34  };
35
36  int main()
37  {
38      A a;
39      B b;
40
41      A* ap = &b;
42      if(ap!=nullptr)
43          ap->print();
44      B* b1 = static_cast<B*> (&a);
45      if(b1!=nullptr)
46          b1->print();
47      B* b2 = static_cast<B*> (ap);
48      if(b2!=nullptr)
49          b2->print();
50      C* c = static_cast<C*> (ap);
51      if(c!=nullptr)
52          c->print();
53
54      A& ar = static_cast<A&> (*ap);
55      ar.print();
56      B& br = static_cast<B&> (*ap);
57      br.print();
58      C& cr = static_cast<C&> (*ap);
59      cr.print();
60
61
62      return 0;
63  }

```

Avem acelasi exemplu ca mai sus, doar ca acum lucram cu static_cast. Sa observam diferentele intre dynamic_cast si static_cast.

Ca si mai sus, la linia 43 se afiseaza 10 7.

Static cast-ul nu face verificari de RTTI, se uita ddoar daca cele 2 tipuri sunt inrudite (in aceeasi ierarhie). Asadar, la linia 44 face cast-ul care ne da un pointer de tip B la un obiect de tip A si la linia 46 se apeleaza print-ul din A, adica se afiseaza 10.

La linia 47 se intampla acelasi lucru ca la exemplul 5, si se afiseaza 10 7.

In schimb la linia 50 se incearca crearea unui pointer de tip C la un obiect de tip B. Am zis ca static cast nu face verificari de RTTI, insa verifica ca tipurile sa fie in aceeasi ierarhie. Cum B si C nu sunt in aceeasi ierarhie, static cast-ul arunca o eroare:

error: invalid static_cast from type 'A*' to type 'C*'

Facem diferenta fata de dynamic cast, care nu arunca eroare la pointeri, doar returna nullptr.

Mai departe la linia 55 se afiseaza 10 7, iar la linia 57 se afiseaza tot 10 7.

In schimb la linia 58 se incearca iar un cast intre tipuri care nu sunt in aceeasi ierarhie, de data asta ca referinte. Avem eroarea:

error: invalid static_cast from type 'A' to type 'C&'

Facem din nou diferenta cu dynamic cast, care ar fi aruncat eroarea std::bad_cast in cazul referintelor.

Ce tip de erori sunt cele de la linia 50 si 58? Erori la compile time. Noi stim de la compile time ca clasele B si C nu se afla in aceeasi ierarhie.

Student la FMI după ce a picat a treia oară examenul la POO

