

Tutoriat 3 POO

Bianca-Mihaela Stan, Silviu Stăncioiu

March 2021



*toate excepțiile au fost tratate în
realizarea acestui album

**Don't know who made this meme,
but good job guys**



1 Object slicing

În tutoriatul trecut, când am făcut despre polimorfism am văzut că putem atribui unui pointer de tipul clasei de bază, un pointer către un obiect de tipul unei clase derivate. În acest fel, aveam acces doar la atributele clasei de baza prin intermediul acelui pointer, dar obiectul pe care îl refeream rămânea la fel.

Dacă nu folosim pointeri, lucrurile se schimbă umpic. Să presupunem că vrem să creem un obiect de tipul clasei de bază care să aibă valoarea unui obiect de tipul clasei derivate. În acest caz, obiectul nou creat va avea toate atributele din clasa de bază a obiectului derivat, dar cele specifice clasei derivate vor fi eliminate.

```
#include <iostream>

using namespace std;

class employee
{
public:
    int yrs_exp;
};

class gamedev : public employee
{
public:
    int games_made;
    int downloads;
};

int main()
{
    employee e = gamedev(); // Aici se
                           // intampla multe
                           // chestii. În
                           // primul rand se
                           // creeaza un
                           // obiect de tipul
                           // gamedev (deci
                           // se apeleaza
                           // mai intai
                           // constructorul
                           // din employee,
                           // iar apoi cel
                           // din gamedev).
                           // Apoi se
                           // taie din obiect
                           // tot ce este
                           // specific lui
                           // gamedev, si
                           // ramane in
                           // obiect doar
                           // variabila
                           // membra yrs_exp.
                           // Acest obiect
                           // taiat este
}
```

```

        // transmis catre
        // copy
        // constructorul
        // din clasa
        // employee,
        // iar apoi obtinem
        // obiectul e, care
        // are doar field
        // -ul yrs_exp.

e.yrs_exp = 10;

        // prin e putem
        // accesa doar
        // variabila
        // yrs_exp deoarece
        // restul au fost
        // taiate.

cout << "Angajatul are " << e.yrs_exp << " ani de experienta." << endl;
cout << "Obiectul are dimensiunea " << sizeof(e) << " bytes"; // se va afisa ca
// obiectul are
// dimensiunea 4
// deoarece
// obiectul e
// contine doar un
// int, iar
// sizeof(int) = 4

return 0;
}

```

După cum se observă, în acest caz, în care nu am folosit pointeri, comportamentul este diferit. Se creează un nou obiect care are doar lucrurile comune cu cel derivat.

Putem observa că object slicing-ul poate duce la comportament neașteptat. Fie următorul exemplu:

```

#include <iostream>

using namespace std;

class employee
{
public:

    virtual void print_type() const
    {
        cout << "the virgin employee" << endl;
    }
};

class gamedev : public employee
{
public:

```

```

virtual void print_type() const override // aceasta suprascriere este specifica
                                         // clasei gamedev.
{
    cout << "the chad gamedev" << endl;
}

};

void print_employee(employee e)
{
    e.print_type();
}

int main()
{
    employee e = employee();
    gamedev g = gamedev();

    print_employee(e);
    print_employee(g);                                // aici se face slicing pentru obiectul
                                                    // g, deci se pierde functia suprascrisa
                                                    // asa ca pentru afisare se va apela
                                                    // functia din clasa employee, deci
                                                    // se va afisa "the virgin employee".

    return 0;
}

```

Dacă lucrăm cu obiecte direct, ci nu cu pointeri, atunci, în acest caz, noi am vrea să modificăm funcția `print_employee` astfel încât aceasta să primească ca argument o referință către obiect. În acest caz, nu s-ar face slicing când am apela `print_employee(g)`, ci am folosi în continuare obiectul `g` aşa cum a fost el creat. Programul modificat astfel încât să fie corect este următorul:

```

#include <iostream>

using namespace std;

class employee
{
public:

    virtual void print_type() const
    {
        cout << "the virgin employee" << endl;
    }
};

class gamedev : public employee
{
public:

    virtual void print_type() const override
    {

```

```

        cout << "the chad gamedev" << endl;
    }

};

void print_employee(employee& e) // acum trimitem o referinta catre obiect, deci nu
                                // se face slicing.
{
    e.print_type();
}

int main()
{
    employee e = employee();
    gamedev g = gamedev();

    print_employee(e);
    print_employee(g);           // va afisa pe ecran "the chad gamedev", deoarece
                                // nu se face slicing, deci obiectul ramane
                                // acelasi, deci se va folosi functia lui
                                // de afisare.

    return 0;
}

```

Alternativa la soluția asta, dacă vrem să nu se facă slicing, este să folosim pointeri, aşa cum am văzut la tutoriatul despre moștenire și polimorfism.

2 Downcasting



Upcasting am vazut cum se poate face în tutoriatul trecut, acum să vorbim puțin despre downcasting. Downcasting este atunci când noi avem un obiect de tipul clasei de bază și ne dorim să îl tratăm ca pe un obiect al clasei derivate.

Dacă noi avem un pointer de tipul clasei de bază către un obiect de tipul clasei derivate, este perfect ok ca noi să transformăm acel pointer într-un pointer al clasei derivate. Nu este nicio problemă în acest caz.

Exemplu:

```
base *b = new derived();
derived *d = (base*)b; // este perfect ok sa schimbam tipul de pointer aici
// deoarece obiectul catre care duce pointerul nostru
// este de tipul derivat, deci il putem folosi
// si ca pe un obiect derivat. Se observa ca am folosit
// (base*) pentru a converti pointerul. Daca nu foloseam
// acest C-style cast, atunci ne-am fi trezit cu o
// eroare de compilare.
```

Totuși, downcasting-ul nu este mereu ceva safe. Dacă în exemplul de mai sus, pointerul *b* nu dusea către un obiect de tipul clasei deriveate, atunci noi ne-am fi trezit cu un program care manipuleaza un obiect de tipul clasei de bază de parcă ar fi un obiect derivat, ceea ce most likely ar fi rezultat în unexpected behaviour, vulnerabilități sau crash-uri.

```
base *b = new base(); // e un obiect de tipul clasei de baza aici
derived *d = (base*)b; // don't do this, guys
```

Cand nu te asiguri că faci downcasting la ce trebuie, dar programul continuă să funcționeze



3 Cast-uri

3.1 C-style cast

C-style cast-ul este cel mai simplu tip de cast, și, aşa cum îi zice numele, era singurul tip de cast din limbajul C. Este cast-ul cu numele tipului de date scris între paranteze înainte de o variabilă. Exemplu:

```
void* ref = (void*)pointer;
```

Acum tip de cast, în cazul pointerilor nu face absolut nimic special. Orice pointer este un număr pe 32 sau 64 de biți, iar acest cast nu îl modifică cu nimic. În cazul în care nu lucrăm cu pointeri, acest tip de cast încearcă să apeleze operatorul de conversie definit în obiect, iar dacă acesta nu există, vom avea o eroare de compilare (excepție se face la object slicing).

3.2 static_cast

static_cast este foarte asemănător cu C-style cast, atât că mai face și câteva verificări la compile time pentru a se asigura că operația de cast are sens. Putem avea de exemplu:

```
derived* d = new derived();
base* b = static_cast<base*>(d);
```

În acest caz, acest cast are sens, compilatorul luându-se după tipul de pointer și observând că d pointează către un obiect al cărui tip moștenește clasa base*. Dacă există o astfel de relație de moștenire, am fi avut o eroare la compilare.

Observație: Dacă tipul de moștenire dintre clase nu este public, *static_cast* va da o eroare la compilare.

Când nu lucrăm cu pointeri, este necesar ca obiectele implicate în cast să aibă operatorii de conversie necesari supraîncărcați pentru a nu primi erori de compilare.

```
#include <iostream>
#include <string>

using namespace std;

class gamedev
{
public:
    gamedev(int yrs_exp)           // avem un constructor care ia
                                    // ca argument un întreg, deci
                                    // putem folosi un întreg
                                    // pentru a crea un gamedev
    {
    }

    operator string()              // aici suprascriem operatorul de
                                    // conversie la string
    {
        return "gamedev, but people call me broke";
    }
};

int main()
{
    gamedev g = static_cast<gamedev>(5); // este ok acest cast, deoarece
                                            // avem constructor care ia ca argument
                                            // un întreg.
    string s = static_cast<string>(g);   // și acest cast este ok, deoarece
                                            // avem un operator de conversie la
                                            // string definit în clasa

    cout << s << endl;                 // se va afisa "gamedev, but people
                                            // call me broke" pe ecran
```

```
    return 0;  
}
```

3.3 dynamic_cast

dynamic_cast este un fel de *static_cast*, doar că mai fancy. *dynamic_cast*, în cazul pointerilor de obiecte nu face verificările la compile time, ci în schimb le face la runtime. Adică în cazul claselor polimorfice se ia după *vptr* pentru a decide dacă se poate face castul. Dacă se poate realiza castul, atunci returnează pointerul de tipul cerut, dacă nu se poate, atunci returnează un pointer null. Exemple:

```
derived *d = new derived();  
base* b = dynamic_cast<base*>(d); // este perfect ok deoarece  
// clasa derived mosteneste clasa  
// base.  
  
#include <iostream>  
  
using namespace std;  
  
class type_a  
{  
    virtual void test() {}  
};  
  
class type_b : public type_a  
{  
};  
  
class type_c : public type_a  
{  
};  
  
int main()  
{  
    type_b b;  
    type_a* ptr = &b;  
    type_c* c = dynamic_cast<type_c*>(ptr);  
}
```

// am adaugat aceasta metoda
// pentru a avea un vptr in clase
// si pentru a avea polimorfism.

// clasa care mosteneste clasa
// type_a

// alta clasa care mosteneste
// clasa type_a

// facem un obiect de tip b
// obtinem o referinta de tip
// type_a catre el

// acest cast nu va da eroare
// de compilare deoarece clasele
// folosite sunt polimorfice
// iar intre ele exista relatii
// de mostenire (este posibil
// sa avem un obiect de tipul
// type_c in pointer de tipul
// type_a, dar trebuie sa
// verificam. Daca pointerul
// era de tipul type_b, atunci
// aveam eroare la compilare).

```

        // Daca aceste
        // lucruri nu erau adevarate, am
        // fi avut erori de compilare.
        // Totusi, va returna un pointer
        // null, deoarece, merand pe
        // vptr, va observa ca tipul
        // obiectului este de fapt
        // type_b, nu type_c, deci
        // castul nu are sens.

    void* afis = (void*)c;

    cout << (int*)(afis) << endl;           // va afisa 0 pe ecran.

    return 0;
}

```

Acet tip de cast merge doar pe pointeri și referințe, altfel va genera erori de compilare.

4 Ascunderea metodelor supraîncărcate

Dacă avem o metodă care este supraîncărcată într-o clasă de bază, iar în clasa derivată este suprascrisă, atunci pentru obiectele de tipul clasei deriveate, supraîncărcările acelei metode vor fi ascunse și nu vor putea fi accesate. Exemplu:

```

#include <iostream>

using namespace std;

class base
{
public:

    void cool()
    {
        cout << "base::cool()";
    }

    void cool(int a)
    {
        cout << "base::cool(int)";
    }
};

class derived : public base
{
public:

    void cool()
    {
        cout << "derived::cool()";
    }
}

```

```
};

int main()
{
    derived d;
    d.cool(10); // functia base::cool(int) este ascunsa
                 // deci avem eroare de compilare.
    return 0;
}
```

Aveti grija ca în practică să nu întâmpinați comportamente de genul. Eu unul nu prea m-am lovit de aşa ceva, dar sunt întrebări la examen cu chestia asta, so.... yeah....

Simbolurile pe care le folosesc eu cand desenez ierarhii sunt:

- + public
- o protected
- private

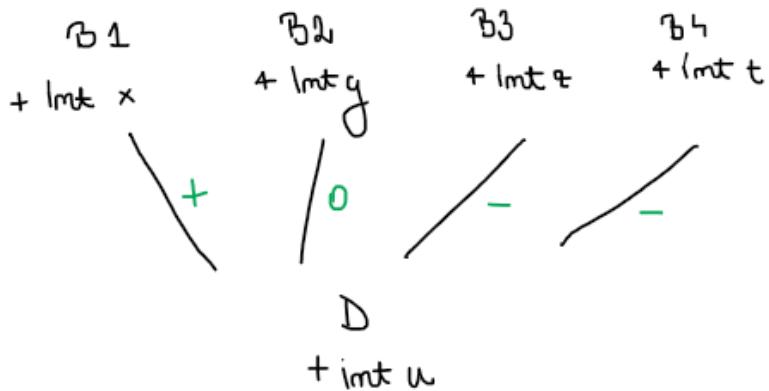
5 Exercitii

Student la FMI după ce vede subiectele de examenul de la POO 2021 color



Exercitiul 1

```
1 #include <iostream>
2 using namespace std;
3 class B1 { public: int x; };
4 class B2 { public: int y; };
5 class B3 { public: int z; };
6 class B4 { public: int t; };
7 class D: public B1, protected B2, B3, private B4
8 {
9     public: int u;
10};
11
12 int main() {
13     D d;
14     cout << d.u;
15     cout << d.x;
16     cout << d.y;
17     cout << d.z;
18     cout << d.t;
19     return 0;
20 }
```



Amintim tabelul:

Access specifier in base class	Access specifier when inherited publicly	Access specifier when inherited privately	Access specifier when inherited protectedly
Public	Public	Private	Protected
Protected	Protected	Private	Protected
Private	Inaccessible	Inaccessible	Inaccessible

Asadar, tot ce era public in clasele de baza devine protected prin mostenire protected (ex: y) si tot ce era public in casa de baza devine private prin mostenire private (ex: z si t).

Eroarea pe care o avem este deci ca y, z si t sunt inaccesibile. Cum reparam? Modificam linia 7:
class D: public B1, public B2, public B3, public B4

Exercitiul 2

```

1 #include <iostream>
2 using namespace std;
3 class A
4 {
5     public:
6         void print() { cout << "A::print()"; }
7     };
8 class B : private A
9 {
10    public:
11        void print() { cout << "B::print()"; }
12    };
13 class C : public B
14 {
15     public:
16         void print() { A::print(); }
17     };
18 int main()
19 {
20     C b;
21     b.print();
22 }
```

A

+ print()



B

+ print()



C

+ print()

Deci print-ul din A, care era public, devine inaccesibil in clasele derivate prin mostenirea privata. Asadar, C nu are acces la print-ul din B. Cum reparam? Modificam linia 8: `class B : public A`

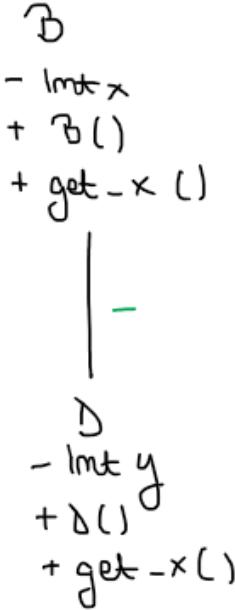
Exercitiul 3

```
#include <iostream>
using namespace std;
class B
{
protected:
    int a;
public:
    B() { a = 7; }
};
class D : public B
{
public:
    int b;
    D() { b = a + 7; }
};
int main()
{
    D d;
    cout << d.b;
    return 0;
}
```

Afiseaza 14.

Exercitiul 4

```
1 #include <iostream>
2 using namespace std;
3 class B
4 {
5     int x;
6 public:
7     B(int v) { v = x; }
8     int get_x() { return x; }
9 };
10 class D : private B
11 {
12     int y;
13 public:
14     D(int v) : B(v) {}
15     int get_x() { return x; }
16 };
17 int main()
18 {
19     D d(10);
20     cout << d.get_x();
21     return 0;
22 }
```



Tot ce era privat in clasa de baza, oricum am mosteni, devine inaccesibil in clasa derivata. Asadar, D-ul nu are acces direct la x-ul din B. Cum reparam? Apelam getter-ul din clasa de baza la linia 15: `int get_x() { return B::get_x(); }`

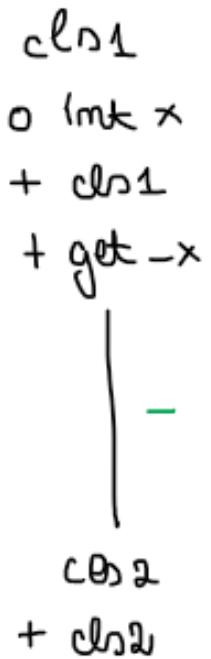
Exercitiul 5

```
1 #include <iostream>
2 using namespace std;
3 class cls1
4 {
```

```

5  protected:
6      int x;
7  public:
8      cls1(int i = 10) { x = i; }
9      int get_x() { return x; }
10 };
11 class cls2 : cls1
12 {
13     public:
14         cls2(int i) : cls1(i) {}
15     };
16 int main()
17 {
18     cls2 d(37);
19     cout << d.get_x();
20     return 0;
21 }

```



Tot ce era public in clasa de baza, prin mostenire privată, devine privat (din tabel). Deci constructorul lui `cls1` si `get_x` sunt metode private in clasa `cls2`. Daca sunt private, inseamna ca nu le putem accesa din afara clasei. Deci nu putem accesa `get_x` la linia 19. Cum rezolvam? Facem mostenirea publica la linia 11: `class cls2 : public cls1`

Exercitiul 6

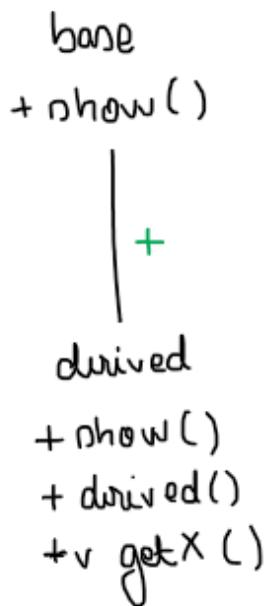
```

1 #include <iostream>
2 using namespace std;
3 class base
4 {
5     public:
6         void show() { cout << " In Base \n"; }

```

```

7  };
8  class derived : public base
9  {
10     int x;
11  public:
12      void show() { cout << "In derived \n"; }
13      derived()
14      {
15         x = 10;
16     }
17     virtual int getX() const { return x; }
18 };
19 int main()
20 {
21     derived d;
22     base *bp = &d;
23     bp->show();
24     cout << bp->getX();
25     return 0;
26 }
```



Fac un obiect de tip derived. Imi iau un pointer de tip baza la el. Acest procedeu se numeste upcasting, pentru ca ma duc in sus pe ierarhie. Cand apelez show() se va afisa In Base. Cand apelez getX am eroare. De ce?

Daca am pointer de tipul bazei, compilatorul crede ca acela este un obiect de tip baza, nu isi da seama ca e de tip derivat. Ca sa faca distinctia la apelarea functiilor, ne-ar trebui functii virtuale.

Pai si puteti spune, getX e deja virtuala. Nu ne ajuta treaba asta, virtual-ul trebuie pus in clasa de baza. Spre exemplu, daca linia 6 o modificam astfel:

```
virtual void show() { cout << " In Base";}  
atunci la linia 23 ar fi afisat In derived.
```

Pentru getX nu avem cum sa punem virtual in baza, pentru ca functia asta nici nu exista in baza. Deci

cum rezolvam problema? Facem pointerul bp de tip derived la linia 22.`derived *bp = d;`

Exercitiul 7

```
#include <iostream>
using namespace std;
class B1
{
public:
    int x;
};
class B2
{
    int y;
};
class B3
{
public:
    int z;
};
class B4
{
public:
    int t;
};
class D : private B1, protected B2, public B3, B4
{
    int u;
};
int main()
{
    D d;
    cout << d.u;
    cout << d.x;
    cout << d.y;
    cout << d.z;
    cout << d.t;
    return 0;
}
```

Nu functioneaza pentru ca u, x, y si t sunt inaccesibili. Analog exercitului 1.

Exercitiul 8

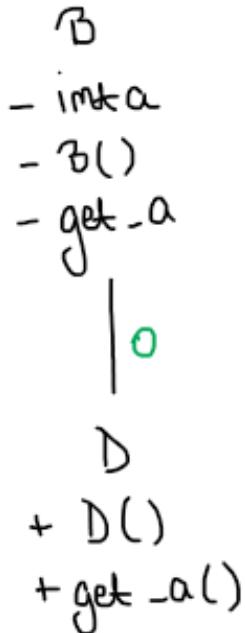
```
#include <iostream>
using namespace std;
class B
{
    int a;
    B(int i = 0) { a = i; }
    int get_a() { return a; }
};
class D : protected B
{
public:
```

```

D(int x = 0) : B(x) {}
    int get_a() { return B::get_a(); }
};

int main()
{
    D d(-89);
    cout << d.get_a();
    return 0;
}

```



Tot ce e privat in clasa de baza, prin orice fel de mostenire am face, devine inaccesibil in clasa derivata. Deci, in derivata nu avem acces nici la constructorul lui B, nici la get_a. Cum rezolvam? Facem datele din clasa de baza publice.

Exercitiul 9

```

#include <iostream>
using namespace std;
class B
{
protected:
    int x;
public:
    B(int i = 0) { x = i; }
};
class D : public B
{
public:
    D() : B(15) {}
    int f() { return x; }
};
int main()
{
    D d;

```

```

    cout << d.f();
    return 0;
}

```

Exercitiul 10

```

#include <iostream>
using namespace std;
class B
{
protected:
    int x;
public:
    B(int i = 16) { x = i; }
    B f(B ob) { return x + ob.x; }
    void afisare() { cout << x; }
};
class D : public B
{
public:
    B f(B ob)
    {
        return x + 1;
    }
};
int main()
{
    B *p1 = new D, *p2 = new B, *p3 = new B(p1->f(*p2));
    p3->afisare();
    return 0;
}

```

Exercitiul 6

```

#include <iostream>
using namespace std;
class A
{
protected:
    int x;
public:
    A(int i = 14) { x = i; }
};
class B : A
{
public:
    B(B &b)
    {
        x = b.x;
    }
    void afisare()
    {
        cout << x;
    }
};

```

```

int main()
{
    B b1, b2(b1);
    b2.afisare();
    return 0;
}

```

Exercitiul 12

```

#include <iostream>
using namespace std;
class B
{
public:
    int x;
    B(int i = 16) { x = i; }
    B f(B ob) { return x + ob.x; }
};
class D : public B
{
public:
    D(int i = 25)
    {
        x = i;
    }
    B f(B ob)
    {
        return x + ob.x + 1;
    }
    void afisare()
    {
        cout << x;
    }
};
int main()
{
    B *p1 = new D, *p2 = new B, *p3 = new B(p1->f(*p2));
    cout << p3->x;
    return 0;
}

```

Exercitiul 13

```

#include <iostream>
using namespace std;
class A
{
    int x;
public:
    A(int i) : x(i) {}
    int get_x() { return x; }
};
class B : public A
{
    int y;

```

```

public:
    B(int i, int j) : y(i), A(j) {}
    int get_y() { return y; }
};

class C : protected B
{
    int z;
public:
    C(int i, int j, int k) : z(i), B(j, k) {}
    int get_z() { return z; }
};

int main()
{
    C c(1, 2, 3);
    cout << c.get_x() + c.get_y() + c.get_z();
    return 0;
}

```

Exercitiul 14

```

#include <iostream>
using namespace std;
class A
{
public:
    int x;
    A(int i = 0)
    {
        x = i;
    }
    A minus()
    {
        return 1 - x;
    }
};

class B : public A
{
public:
    B(int i = 0) { x = i; }
    void afisare() { cout << x; }
};

int main()
{
    A *p1 = new B(18);
    *p1 = p1->minus();
    p1->afisare();
    return 0;
}

```

Exercitiul 15

```

#include <iostream>
using namespace std;
class A
{

```

```

protected:
    int x;
public:
    A(int i = -16) { x = i; }
    A f(A a) { return x + a.x; }
    void afisare() { cout << x; }
};

class B : public A
{
public:
    B(int i = 3) : A(i) {}
    B f(B b) { return x + b.x + 1; }
};

int main()
{
    A *p1 = new B, *p2 = new A, *p3 = new A(p1->f(*p2));
    p3->afisare();
    return 0;
}

```

Exercitiul 16

```

#include <iostream>
using namespace std;
class B
{
    int i;
public:
    B() { i = 1; }
    int get_i() { return i; }
};

class D : public B
{
    int j;
public:
    D() { j = 2; }
    int get_i() { return B::get_i() + j; }
};

int main()
{
    const int i = cin.get();
    if (i % 3)
    {
        D o;
    }
    else
    {
        B o;
    }
    cout << o.get_i();
    return 0;
}

```

Exercitiul 17

```

#include <iostream>
using namespace std;
class B
{
protected:
    int x;
public:
    B(int i = 28) { x = i; }
    B f(B ob)
    {
        cout<<"B\n";
        return x + ob.x + 1;
    }
    void afisare() { cout << x; }
};
class D : public B
{
public:
    D(int i = -32) : B(i) {}
    B f(B ob)
    {
        cout<<"D\n";
        return x + ob.x - 1;
    }
};
int main()
{
    B *p1 = new D, *p2 = new B, *p3 = new B(p1->f(*p2));
    p3->afisare();
    return 0;
}

```

Exercitiul 18

```

#include <iostream>
using namespace std;
class A
{
public:
    int x;
    A(int i = 0) { x = i; }
    virtual A minus() { return (1 - x); }
};
class B : public A
{
public:
    B(int i = 0) { x = i; }
    void afisare() { cout << x; }
};
int main()
{
    A *p1 = new B(18);
    *p1 = p1->minus();
    p1->afisare();
}

```

```
    return 0;
}
```

Exercitiul 19

```
#include <iostream>
using namespace std;
class A
{
public:
    int x;
    A(int i = 0) { x = i; }
    virtual A minus() { cout << x; }
};
class B : public A
{
public:
    B(int i = 0) { x = i; }
    void afisare() { cout << x; }
};
int main()
{
    A *p1 = new A(18);
    *p1 = p1->minus();
    dynamic_cast<B *>(p1)->afisare();
    return 0;
}
```

Exercitiul 20

```
#include <iostream>
using namespace std;
class A
{
public:
    virtual void fun() { cout << "A" << endl; }
};
class B : public A
{
public:
    virtual void fun() { cout << "B" << endl; }
};
class C : public B
{
public:
    virtual void fun() { cout << "C" << endl; }
};
int main()
{
    A *a = new C;
    A *b = new B;
    a->fun();
    b->fun();
    return 0;
}
```

Exercitiul 21

```
#include <iostream>
#include <stdio.h>
using namespace std;
class Base
{
public:
    Base()
    {
        fun();
    }
    virtual void fun()
    {
        cout << "\nBase Function";
    }
};
class Derived : public Base
{
public:
    Derived() {}
    virtual void fun()
    {
        cout << "\nDerived Function";
    }
};
int main()
{
    Base *pBase = new Derived();
    delete pBase;
    return 0;
}
```

Exercitiul 22

```
#include <iostream>
using namespace std;
class base
{
public:
    virtual void show() { cout << " In Base \n"; }
};
class derived : public base
{
    int x;
public:
    void show() { cout << "In derived \n"; }
    derived() { x = 10; }
    int getX() const { return x; }
};
int main()
{
    derived d;
    base *bp = &d;
    bp->show();
```

```
    cout << bp->getX();
    return 0;
}
```

Exercitiul 23

```
#include <iostream>
using namespace std;
class A
{
protected:
    int x;
public:
    A(int i = -31) { x = i; }
    virtual A operator+(A a) { return x + a.x; }
};
class B : public A
{
public:
    B(int i = 12) { x = i; }
    B operator+(B b) { return x + b.x + 1; }
    void afisare() { cout << x; }
};
int main()
{
    A *p1 = new B, *p2 = new A;
    B *p3 = new A(p2->operator+(*p1));
    p3->afisare();
    return 0;
}
```

