Tutoriat 2 POO

Bianca-Mihaela Stan, Silviu Stăncioiu

March 2021

Awww! Thx so much! You're breathtaking!

Urare basic scrisa pe wall

Urare pe un meme template overly-used: La mulți ani celui mai smecher tutore!! 🚙🌆✨

1 Moștenirea

Până acum am avut clase total independente care se ocupau de un singur lucru. Totuși, se poate întâmpla să avem nevoie de clase cu funcționalități similare, chiar dacă sunt clase diferite.

Presupunem că avem o clasă care se ocupă cu gestionarea unui angajat. To keep things simple, clasa noastră va avea un singur rol, acela de a calcula salariul unui angajat. Putem face o simplă clasă care să aibă doar un getter pentru salariu. O astfel de clasă ar arăta așa:

```
class employee
{
public:

    employee(int salary) :
        _salary(salary)
    {
    }

    int get_salary() { return _salary; }

private:

    int _salary;
}
```

Clasa asta este ok pentru un tip de angajat basic să zicem, totuși în lumea reală, salariul unui angajat depinde foarte multe de profesia acestuia. Să presupunem că vrem să aflăm salariile pentru doi angajați cu meserii diferite. Presupunem că formula de calcul pentru ei este diferită. În acest caz putem avea următorul cod pentru a ne calcula salariul unui gamedev și al unui webdev:

```
#include <iostream>
```

```
using namespace std;

enum employee_type                                  // avem o enumeratie
                                                    // acesta este un tip de
                                                    // date care poate
                                                    // avea doar valorile
                                                    // declarate in acolada
                                                    // adica gamedev si
                                                    // webdev.
                                                    // fiecare element
                                                    // declarat in acolade
                                                    // are atribuit un număr
                                                    // (gamedev are numarul
                                                    // 0, iar webdev are
                                                    // numarul 1). Deci
                                                    // ((int)webdev ==
                                                    // 1) va returna true.
{
    gamedev,
    webdev
};

class employee
{
public:

    employee(employee_type emp, int yrs_exp) :      // un constructor basic
                                                    // care doar salveaza
                                                    // tipul de angajat
                                                    // si numarul de ani
                                                    // de experienta
                                                    // ai acestuia.
        _emp_type(emp),
        _yrs_exp(yrs_exp)
    {
    }

    int get_salary()                                // functia care ne
                                                    // returneaza salariul
                                                    // in functie de tipul
                                                    // de angajat
    {
        switch (_emp_type)
        {
            case employee_type::gamedev:            // avem un gamedev, deci
                                                    // returnam salariul
                                                    // unui gamedev.

                                                    // se observa ca am
                                                    // folosit
                                                    // employee_type::gamedev
                                                    // pentru a verifica valoarea
                                                    // lui _emp_type. Puteam
```

```
                                                // scrie direct gamedev acolo.

                return 0;                       // gamedevii nu merita
                                                // sa fie platiti.
                                                // if you want to donate
                                                // though, DM me. I like
                                                // DOGE coins!!!

                break;                          // acest break este
                                                // inutil inutil
                                                // deoarece se da return
                                                // inainte. A fost
                                                // adaugat doar in scop
                                                // didactic pentru
                                                // a nu uita ca se pune
                                                // break dupa case-urile
                                                // din switch. In real
                                                // life coding as fi
                                                // omis acest break.
            case employee_type::webdev:         // aici vom returna
                                                // salariul unui webdev

                return min(_yrs_exp * 1000, 3000);
                break;                          // again, break-ul asta
                                                // este inutil.
        }

        return -1;                              // nu se va ajunge
                                                // aici niciodata,
                                                // deoarece am tratat
                                                // toate cazurile.
                                                // Totusi, daca mai
                                                // adaugam un caz si
                                                // uitam sa il tratam,
                                                // va ajunge aici.
    }

private:

    employee_type _emp_type;                    // variabila de tipul
                                                // employee_type care
                                                // ne spune ce tip
                                                // de angajat avem.
    int           _yrs_exp;
};

int main()
{
    employee gamedev(employee_type::gamedev, 10); // facem un gamedev
                                                  // cu 10 ani de
                                                  // experienta

    employee webdev(employee_type::webdev, 3);    // facem un webdev cu
                                                  // 3 ani de experienta
```

```
                                                    // afisam salariile celor
                                                    // doi angajati.

    cout << "Salariul unui gamedev este: " << gamedev.get_salary() << endl;
    cout << "Salariul unui webdev este: " << webdev.get_salary() << endl;

    return 0;
}
```

Observație

Am folosit un enum. Un enum ne permite sa creem tipuri noi de date care pot avea doar câteva doar câteva valori, definite de noi. În cazul de mai sus, valorile pe care le poate lua o variabilă de tipul *employee_type* sunt doar *gamedev* și *webdev*. Pentru a specifica o valoare de tipul *employee_type* se poate scrie *employee_type* :: *gamedev* sau direct *gamedev* (la fel și pentru *webdev*). Valorile dintr-un enum sunt de fapt niște numere (*gamedev* e 0, *webdev* e 1, and so on...). Se pot suprascrie numrele atribuite valorilor în felul următor:

```
enum employee_type
{
    gamedev = 1,
    webdev = 0
};
```

Aici am făcut ca valoarea lui *gamedev* să fie 1, iar valoarea lui *webdev* să fie 0. Este foarte util ca aceste valori să fie de fapt numere, pentru că putem face cast-uri între *int* și tipuri definite cu *enum*. Când facem *sizeof* unui tip de date declarat cu enum depinde foarte mult de de numerele atribuite valorilor câți bytes sunt necesari în reprezentarea lor binară.

```
#include <iostream>

using namespace std;

enum foo
{
    rick = 0
};

enum bar
{
    roll = 1000000000000
};

int main()
{
    foo a = foo::rick;
    bar b = bar::roll;
    cout << sizeof(a) << endl << sizeof(b); // va afisa pe primul rand 4
                                            // pe al doilea rand va afisa
                                            // 8.

    return 0;
}
```

Ok, back to business. Soluția de mai sus în care avem o clasă *employee* care se ocupă și de *gamedevi* și de *webdevi* nu este o soluție foarte inspirată, deoarece de ficare dată când mai adăugăm un tip de angajat, este nevoie să ne asigurăm că noi încă avem o clasă care funcționează. Altă soluție ar fi fost să facem două clase, una pentru *gamedev* și cealaltă pentru *webdev*. Totuși, nici în acest caz nu ar fi fost ok, pentru că am fi avut cod redundant (ambele ar fi avut funcționalități comune). Exemplu:

```
#include <iostream>

using namespace std;

class gamedev
{
public:

    gamedev(int yrs_exp) :
        _yrs_exp(yrs_exp)
    {
    }

    int get_salary()
    {
        return 0 * _yrs_exp; // inmultit cu _yrs_exp doar ca sa nu zic
                             // ca am pus _yrs_exp in clasa degeaba (desi
                             // e degeaba). E un exemplu didactic....
    }

private:

    int _yrs_exp;
};

class webdev
{
public:

    webdev(int yrs_exp) :    // primul lucru reduntant.. avem de doua ori
                             // un constructor care primeste numarul de
                             // ani de experienta. O data in gamedev
                             // si o data in webdev.
        _yrs_exp(yrs_exp)
    {
    }

    int get_salary()         // si aici avem redundanta
                             // pentru ca avem o functie
                             // care face acelasi lucru
                             // in doua clase.
                             // valoarea returnata e diferita tho'..
    {
        return min(1000 * _yrs_exp, 3000);
    }

private:

    int _yrs_exp;            // si aici este reduntanta...
```

```
                                // aceasta variabilă a fost declarată
                                // și în gamedev și în webdev.
};

int main()
{
    gamedev g(10);
    webdev w(3);

    cout << g.get_salary() << endl;
    cout << w.get_salary() << endl;

    return 0;
}
```

Ok, deci dacă facem o clasă de uz general nu e bine că trebuie să modificăm mereu când adăugăm un nou tip de angajat. Iar când avem două clase independente nu e bine pentru că avem redundanță... Atunci ce ne faceeeeeeeeem???????

Well.... moștenire, lol. Ideea e că putem avea o clasă de bază *employee* care să conțină doar caracteristicile comune gamedevilor și webdevilor, iar apoi două clase, una pentru *gamedev* și cealaltă pentru *webdev* care să conțină doar lucrurile specifice celor două meserii. Prin moștenire, vom face ca ambele clase să aibă caracteristiile clasei *employee*. Avem codul:

```
#include <iostream>

using namespace std;

class employee                          // clasa noastră cu atributele
                                        // comune
                                        // tuturor meseriilor
{
public:

    employee(int yrs_exp) :             // pentru orice meserie avem
                                        // nevoie de un număr de ani de
                                        // experiență. deci vom face un
                                        // constructor care primeste
                                        // numarul de ani de experienta ai
                                        // unul angajat.
        _yrs_exp(yrs_exp)               // salvam numarul de ani de de
                                        // experienta intr-o variabila
                                        // memebra
    {
    }

    int get_salary()
    {
        return -1;                      // nu stim ce fel de angajat avem,
                                        // deci vom returna -1.
    }

protected:
```

```
    int _yrs_exp;                       // avem o variabilă de tip
                                        // protected care stocheaza
                                        // numarul de ani de experienta.
                                        // vom reveni mai tarziu cu
                                        // detalii despre protected
                                        // momentan trebuie sa stim ca
                                        // o valoare protected poate fi
                                        // accesata doar de clasele care
                                        // mostenesc clasa aceasta.
                                        // ea nu poate fi accesata
                                        // din exterior. Este cumva
                                        // la mijloc intre public si
                                        // private (variabilele private nu
                                        // pot fi accesate nici din
                                        // clasele care mostenesc clasa
                                        // aceasta)
};

class gamedev : public employee         // aici ii spunem faptul ca avem
                                        // o clasa care mosteneste
                                        // proprietatile clasei employee.
                                        // In acest caz mostenirea
                                        // este publica (vom vedea mai
                                        // incolo ce inseamna asta).
{
public:

    gamedev(int yrs_exp) :              // constrcuctorul clasei gamedev,
                                        // cere numarul de ani de
                                        // experienta (... nu ca ar avea
                                        // vreo relevanta, un gamedev nu
                                        // este platit)

        employee(yrs_exp)               // din moment ce noi mostenim
                                        // clasa employee,  atunci cand
                                        // creem un obiect care o
                                        // mosteneste, trebuie sa ii
                                        // apelam constructorul inainte de
                                        // a apela constructorul din
                                        // obiectul care o mosteneste.
                                        // Facem acest lucru in interiorul
                                        // listei de initializare.
    {
    }

    int get_salary()
    {
        return 0;                       // :'(
    }
};

class webdev : public employee          // din nou, creem o clasa care
                                        // mosteneste clasa employee
{
```

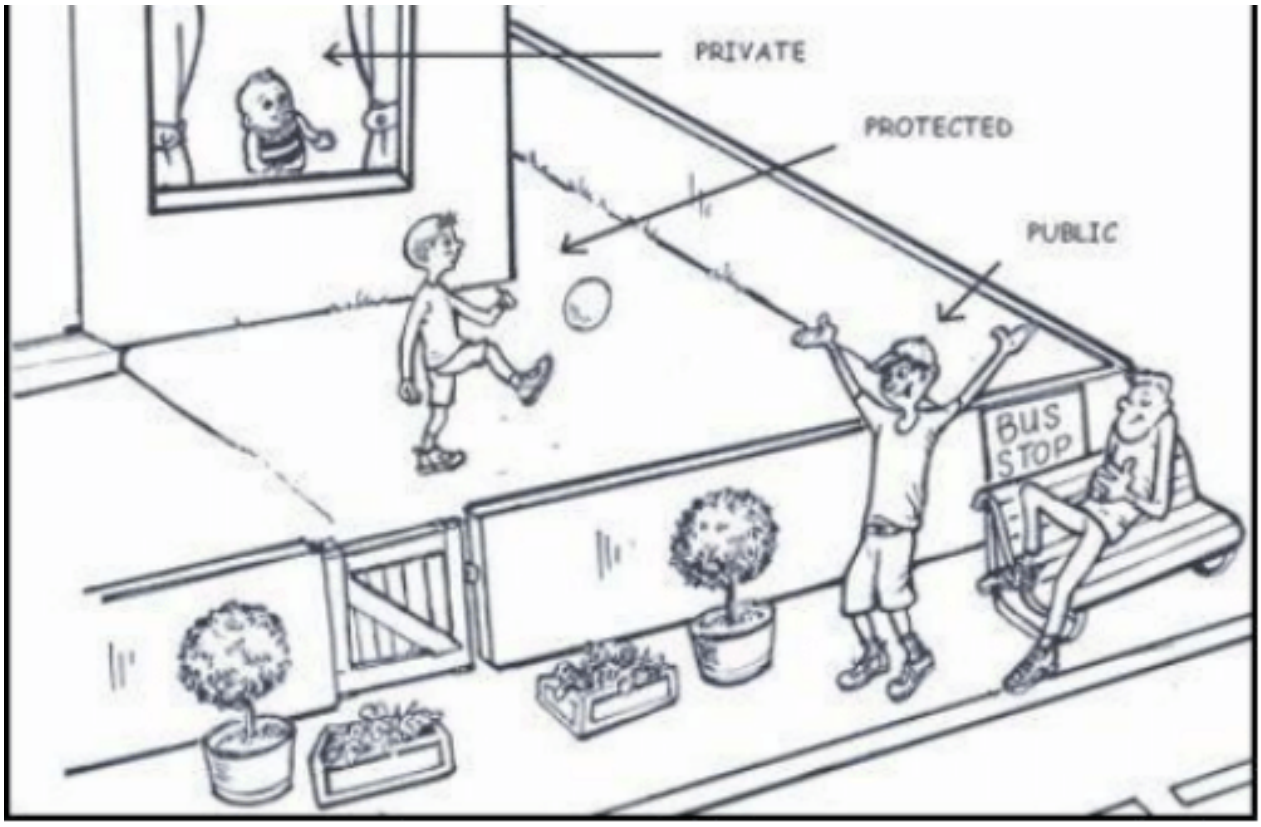
```
public:

    webdev(int yrs_exp) :
        employee(yrs_exp)                   // again, apelam constructorul din
                                            // employee in lista de
                                            // initializare din clasa noua.
    {
    }

    int get_salary()
    {
        return min(_yrs_exp * 1000, 3000); // se observa ca avem acces la
                                            // variabila membra _yrs_exp
                                            // deoarece aceasta este protected
                                            // in clasa employee, deci
                                            // in clasele care mostenesc
                                            // clasa employee avem acces la
                                            // ea. Daca variabila era private
                                            // noi nu am fi avut acces la ea
                                            // aici.
    }
};

int main()
{
    gamedev g(10);                          // de aici in jos
                                            // este fix ca in exemplul
                                            // cu doua clase de mai sus.
    webdev w(3);

    cout << g.get_salary() << endl;
    cout << w.get_salary() << endl;

    return 0;
}
```

De acum, de fiecare dată când vrem să mai adăugăm un nou tip de angajat în sistem, trebuie doar sa scriem o clasă pentru el, clasă care să moștenească clasa employee. Momentan exemplul nostru este banal, dar într-o aplicație mai mare, este foarte probabil ca angajații să aibă multe proprietăți în comun, nu doar numărul de ani de experiență. În acest caz, o metodă foarte elegantă este să folosim moștenire.

Să ne reamintim puțin specificatorii de acces (din tutoriatul trecut). Avem:

- *public*, ne permite sa accesam proprietati ale clasei din orice parte a codului. Cat timp avem un obiect, putem accesa orice are acesta public, oriunde ne-am afla în cod. Again, nu e recomandat la examenul de POO sa aveti date membre publice (restanta). Tbh, în practica se mai folosesc date membre publice (mai ales in structuri pentru headere de fisiere), but hey, I didn't make the rules for this exam.
- *protected*, este mai putin permisiv decât *public*. Ne permite să accesăm proprietățile doar din interiorul clasei sau din interiorul claselor care moștenesc clasa respectivă. (sau claselor care mostenesc clasa mostenita... and so on... bine... aici depinde de tipul de mostenire, dar vorbim despre asta mai incolo).
- *private*, este cel mai restrictiv dintre toate. By default, orice se află într-o clasă este *private* (în afara de constructorul implicit), dacă nu se specifică. Datele private pot fi accesate doar din interiorul clasei, nu pot fi accesate în altă parte a codului (excepție se poate face în cazul claselor $frined...$ $friend$ se învață la POO, dar nu este considerat un good practice).


PRIVATE
PUBLIC
PROTECTED
Allow us to introduce ourselves,



Perfect analogy for the variable data types in Object-Oriented Programming

1.1 Tipuri de moștenire

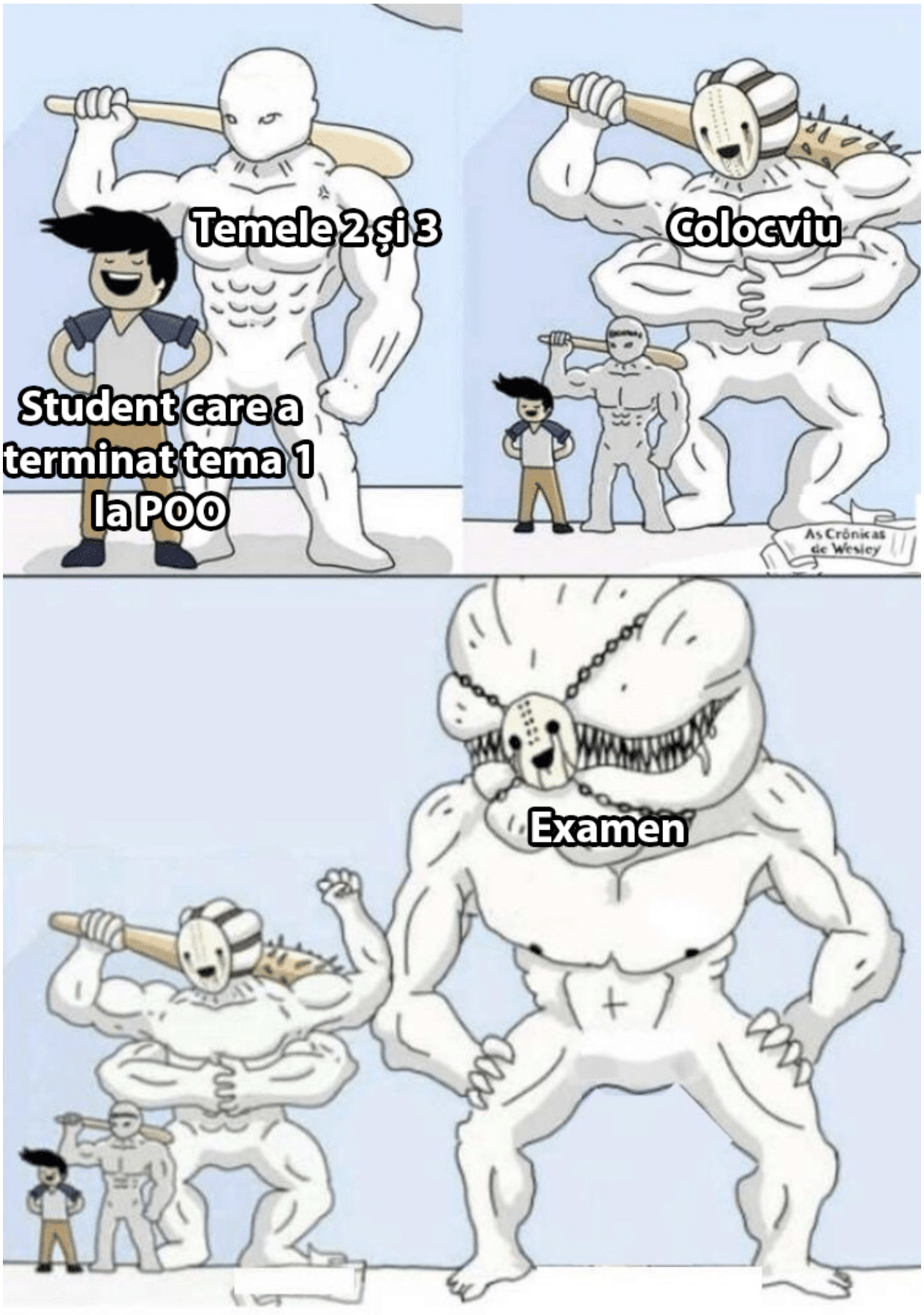
Ne-am amintit mai sus specificatorii de acces. Se observă că în exemplul de mai sus cu moștenire, atunci când am creat clasele gamedev si webdev, după semnul : am scris *public employee*. Adică tipul de moștenire să fie *public*. Moștenirea poate fi de 3 tipuri: *public*, *protected* și *private*. Dacă nu se specifică tipul de moștenire, atunci, by default, avem moștenire *private*.

- Moștenire *public* - acesta este cel mai întâlnit tip de moștenire, și după părerea mea, celelalte două sunt cam inutile, but we'll cover them too. Ideea e că atunci când avem o clasă care moștenește public altă clasă, toate lucrurile publice din clasa de baza rămân publice și în clasa derivată, toate lucrurile protected din clasa de bază devin protected și în clasa derivată, iar toate lucrurile private din clasa de baza, rămân private și pentru clasa derivată, deci clasa derivată nu are acces la ele.
- Moștenire *protected* - această moștenire se comportă în felul următor: lucrurile publice din clasa de bază se transformă în protected în clasa derivată, lucrurile protected din clasa de bază rămân protected și în clasa derivată, iar lucrurile private din clasa de bază, rămân private și în clasa derivată, deci clasa derivată nu are acces la ele.

- Moștenire *private* - toate lucrurile din clasa de bază devin private în clasa moștenită, deci nu avem acces la ele.

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

See the pattern? Fiecare tip de moștenire face ca în clasa derivată, lucrurile din clasa de bază să fie cel mult tipul moștenrii. La moștenirea public, lucrurile moștenite în clasa derivată sunt cel mult publice, la moștenirea protected lucrurile moștenite în clasa derivată sunt cel mult protected (public se transformă în protected), iar la cea private, toate lucrurile devin private.


Temele 2 și 3
Student care a terminat tema 1 la POO
Colocviu
As Crônicas de Wesley
Examen

2 Polimorfismul

Polimorfismul in C++ vine in 2 forme:

- polimorfismul la compilare (compile time)
- polimorfismul la executie (run time)

In primul rand, ce sunt compile time si run time?

- La compile time se fac verificari de sintaxa (daca sunt puse bine ; , daca s-a scris un if folosind sintaxa corecta, etc) si verificari de semantica. Ce inseamna asta?

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

void f (char* x) {}

int main()
{
    int x;                  // de acum incolo, cand spun x,
                            // programul meu stie ca ma refer
                            // la variabila x de tip int

    f(x);                   // daca acum il trimit pe x ca
                            // ca parametru care primeste un
                            // char*, voi primi o eraore la
                            // compilare:
                            // argument of type "int" is incompatible
                            //  with parameter of type "char *"
    return 0;
}
```

- La run time se aloca memoria necesara variabilelor. Erori pe care le putem intalni la run time sunt:
  - impartire la 0
  - dereferentierea unui pointer null
  - am ramas fara memorie

2.1 Polimorfismul la compilare

Cum se realizeaza polimorfismul la compilare? Prin supraincarcare de functii si de operatori.

2.1.1 Supraincarcarea functiilor

Acum ca am ajuns la subiectul acesta, ar fi bine sa ne uitam la anumite detalii referitoare la supraincarcarea functiilor.

Atunci and aplez o functie, compilatorul cauta, in ordinea aceasta:

1. O potrivire exacta a parametrilor.

```
#include <iostream>
using namespace std;
void f(char* x);
void f(int value); // se alege asta ca e potrivire exacta

int main()
```

```
{
    f(0);
    return 0;
}
```

2. Daca nu se gaseste o potrivire exacta, se trece se trece la promovarea prin internal type conversion. Exita anumire tipuri in C++ care pot fi promovate automat la alte tipuri prin internal type conversion:

   - char, unsigned char si short sunt promovate la int
   - unsigned short poate fi promovat la unsigned int sau la int
   - float poate fi promovat la double
   - enum poate fi promovat la int

```
#include <iostream>
using namespace std;
void f(char* x);
void f(int x);

int main()
{
    f('a'); // char-ul e promovat la int
    return 0;
}
```

3. Se incearca apoi conversia prin standard conversion:

   - orice tip numeric va fi potrivit cu orice alt tip numeric.
   - enum va fi potrivit cu tipurile numerice (Ex: enum la float)
   - 0 se va potrivi cu pointeri si tipuri numerice(ex: 0 la char* si 0 la float)
   - un pointer se va potriuvi cu void*

```
#include <iostream>
using namespace std;
struct employee;
void f(float x);
void f(employee x);

int main()
{
    f('a'); // 'a' se converteste la int, apoi la float
    return 0;
}
```

4. Se inceraca potrivirea prin conversie definita de utilizator.

```
#include <iostream>
using namespace std;
class A
{
public:
    operator int()// operator de conversie definit de noi
    {
        return 9;
    }
};
```

```
void f(float x);
void f(int x);

int main()
{
    A a;
    f(a); // a se converteste la int prin operatorul int()
    return 0;
}
```

5. Daca nu se gaseste niciuna dintre astea, e eroare.

Potrivirea ambigua este atunci cand avem mai multe potriviri, pe acelasi nivel (nivelele 1, 2, 3 sau 4 enumerate mai sus).

Exemplu:

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

void f(unsigned int x);
void f(float x);

int main()
{
    f('a');                 // Nivelul 1: nu exista potrivire perfecta.
                            // Nivelul 2: se converteste char la un int,
                            // dar nu exista niciun f(int).
                            // Nivelul 3: un char se poate conevrti si la
                            // float si la unsigned int.
                            // Deci avem mai multe potriviri la acelasi
                            // nivel.
                            // => eroare: more than one instance of
                            // overloaded function "f" matches the
                            // argument list
    return 0;
}
```

Supraincarcarea functiilor pentru functii cu mai multi parametri Se alege functia care ofera o potrivire mai exacta pe un parametru si cel putin la fel de exacta ca celelalte functii pentru ceilalti parametri.

Exemplu:

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

void f(char c, int x); // se alege asta
void f(char c, double x);
void f(char c, float x);

int main()
{
    f('a', 4);                  // se alege acea functie pentru ca
                                // ne da o potrivire mai exacta pentru 4
                                // si o potrivire cel putin la fel de buna
                                // ca a celorlalte functii pentru 'a'
    return 0;
}
```

2.1.2 Supraincarcarea operatorilor

Cateva reguli de baza:

- Daca aveti de suprainacrcat oepartorii =, [], () sau − >, supraincarcarea se face ca **functie membra a clasei**.
- Daca aveti de supraincarcat un operator unar, supraincarcati-l ca **functie membra a clasei**.
- Daca aveti de supraincarcat un operator binar:
  - Daca operatorul nu modifica termenul din stanga (exemplu: operatorul +) supraincarcati ca **functie normala** (de preferat) sau **functie prietena**.
  - Daca operatorul modifica termenul din stanga, dar nu putem modifica definitia acelui termen (exemplu: operatorul << care are termenul din stanga de tip ostream), supraincarcati ca **functie normala** (de preferat) sau functie friend.
  - Daca operatorul modifica termenul din stanga (exemplu: operatorul+=) si se poate modifica definitia termenului din stanga, supraincarcati ca **functie membra a clasei**.

Link-uri utile:

- Supraincarcarea operatorilor aritmetici prin functii friend
- Supraincarea operatorilor prin functii normale
- Supraincarcarea operatorilor folosind functii membre
- Supraincarcarea operatorilor unari +, -, !
- Supraincarcarea operatorilor de comparatie
- Supraincarcarea operatorilor ++ si - -
- Supraincarcarea operatorului []
- Supraincarcarea operatorului ()

2.1.3 Operatorul=

Regula celor 3 spune ca, daca supraincarcam unul dintre aceste 3 chestii, trebuie sa le supraincarcam pe toate 3: constructorul, constructorul de copiere si operatorul=.

Operatorul= si constructorul de copiere fac cam acelasi lucru: amandoua copiaza un obiect in altul. Diferenta este data de faptul ca la operatorul= obiectul deja exista, pe cand copy constructorul il construieste atunci.

Cum supraincarcam operatorul= ?

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

class example
{
public:
    // Primeste ca parametru const example& din aceleasi motive
    // pentru care primea si constructorul de copiere.
    // Pentru mai multe detalii vezi Tutoriat 1 : paginile 30-32.
    example& operator= (const example& obj)
    {
        delete[] _a;                // Principala diferenta fata de
```

```
                                    // copy constructor este ca trebuie
                                    // sa ne asiguram ca dezalocam memoria
                                    // alocata inainte, pentru ca acest obiect
                                    // era deja creat.
                                    // Daca nu as dezaloca acum memoria as
                                    // pierde acest pointer si dupa
                                    // nici daca as vrea nu as mai putea
                                    // dezaloca. => memory leak
        _n=obj._n;
        _a=new int[_n];
        for(int i=0; i<_n; i++)
        {
            _a[i]=obj._a[i];
        }

        return *this; // returnez obiectul care a apelat operatorul=
    }
    // pun si copy constructorul aici ca sa vedeti diferentele
    example(const example& obj)
    {
        _n=obj._n;
        _a=new int[_n];
        for(int i=0; i<_n; i++)
        {
            _a[i]=obj._a[i];
        }
    }

    example(int n=0)
    {
        _n=n;
        _a=new int[_n];
    }
private:
    int _n;
    int* _a;
};

int main()
{
    example ob1;
    example ob2;
    return 0;
}
```

Atentie! Operatorul= nu poate fi supraincarcat ca functie friend. De ce? Pentru ca asa s-a stabilit printr-un standard. Operatorul= este semantically bound to the left hand side, adica modifica argumentul din stanga ca sa fie egal cu argumentul din dreapta. Alti algoritmi, cum ar fi +, sunt mai flexibili. Ce inseamna a+b? Poate sa insemne si modifica a in a+b si modifica b in a+b.

2.1.4 Operatorii >> si <<

Operatorii >> si << pot fi supraincarcati doar ca functii friend. Cum se supraincarca?

```
#include <iostream>
#include <ctime>
```

```
using namespace std;

class punct2D
{
public:
    friend std::ostream& operator<< (std::ostream& out, const punct2D& ob);
    friend std::istream& operator>> (std::istream& in,  punct2D& ob);
private:
    int _x;
    int _y;
};
std::ostream& operator<< (std::ostream& out, const punct2D& ob)
{
    out<<"("<<ob._x<<", "<<ob._y<<")";
}

std::istream& operator>> (std::istream& in,  punct2D& ob)
{
    in>>ob._x;
    in>>ob._y;
}

class array_de_puncte2D
{
public:
    friend std::ostream& operator<< (std::ostream& out, const array_de_puncte2D& ob);
    friend std::istream& operator>> (std::istream& in,  array_de_puncte2D& ob);
private:
    int _n;
    punct2D* _arr;
};

// In operatorii de input si output de la array_de_puncte2D ne folosim de
// operatorii definiti pentru punct2D. Cum? (***)
std::ostream& operator<< (std::ostream& out, const array_de_puncte2D& ob)
{
    for(int i=0; i<ob._n; i++)
    {
        // (***) aici: se apeleaza operatorul de output pentru un element de tip punct2D
        out<<ob._arr[i]<<" ";
    }
}

std::istream& operator>> (std::istream& in,  array_de_puncte2D& ob)
{
    // Nu se recomanda afisarea(cout-ul) in cadrul operatorului de input, insa
    // cat timp faceti debugging eu zic ca e util ca sa vedeti mai usor erorile.
    // Dupa ce va asigurati ca merge e bine sa stergeti cout-urile ca sa nu va depuncteze.
    cout<<"Introduceti numarul de elemente al array-ului: ";
    in>>ob._n;
    ob._arr=new punct2D[ob._n];
    cout<<"\n";

    for(int i=0; i<ob._n; i++)
```

```
    {
        cout<<"Introduceti un element: ";
        // (***) aici: se apeleaza operatorul de input pentru un element de tip punct2D
        in>>ob._arr[i];
        cout<<"\n";
    }
}

int main()
{
    return 0;
}
```

2.1.5 Operatorul de typecast

Operatorul de typecast ne permite sa convertim tipul nostru de date la un alt tip.

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

class portofel
{
public:
    portofel(int nr_monede=0) : _nr_monede(nr_monede) {};

    //operator de typecast la int
    operator int() const { return _nr_monede; }
    //operator de typecast la string
    operator string() const { return to_string(_nr_monede) + " monede";}

    int get_nr_monede() const { return _nr_monede; }
    int set_nr_monede(int nr_monede) { _nr_monede=nr_monede; }
private:
    int _nr_monede;
};

int main()
{
    portofel ob1(9);
    cout<<int(ob1)<<"\n";           // afiseaza 9
    cout<<string(ob1);              // afiseaza 9 monede
    return 0;
}
```

2.2 Polimorfismul la executie

Sa intelegem mai intai de ce avem nevoie de polimorfismul la executie.

Avem 3 clase: `animal`, `cat` si `dog`. Vrem sa tinem pointeri de tip `animal` la `dog` si `cat`.

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

class animal
{
```

```
public:
    void sound() {cout<<"undefined \n";}
};

class cat : public animal
{
public:
    void sound() {cout<<"meow \n";}
};

class dog : public animal
{
public:
    void sound() {cout<<"wof \n";}
};


int main()
{
    cat cat1;
    cat1.sound();       // se afiseaza meow

    dog dog1;
    dog1.sound();       // se afiseaza wof

    cat* catp=new cat;
    catp->sound();      // se afiseaza meow

    dog* dogp=new dog;
    dogp->sound();      // se afiseaza wof

    // Putem sa facem un animal* pointer catre un obiect de tip cat sau dog.
    // Putem, pentru ca fiecare cat sau dog are o parte de animal.

    animal* animal1 = new cat;
    animal* animal2 = new dog;

    animal1->sound();   // se afiseaza undefined
    animal2->sound();   // se afiseaza undefined

    return 0;
}
```

Well, that's a bummer. Fiindca avem un `animal*` la `cat`, se apeleaza functia sound din `animal`. Acest mic bummber are implicatii mai mare totusi. De multe ori vom avea astfel de ierarhii cu o clasa de baza mai generala cum este `animal`. Vom vrea sa tinem vectori de pointeri la astfel de obiecte si sa apelam o anumita functie din fiecare element in parte. Cum facem ca acea functie apelata sa fie cea din `cat` sau `dog` si nu `animal`?

Aici intervine polimorfismul la executie si functiile virtuale.

Putem rezolva problema de mai sus prin adaugarea keyword-ului `virtual` la functia `sound` din clasa de baza `animal`:

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

```
class animal
{
public:
    virtual void sound() {cout<<"undefined \n";}
};

class cat : public animal
{
public:
    void sound() {cout<<"meow \n";}
};

class dog : public animal
{
public:
    void sound() {cout<<"wof \n";}
};


int main()
{
    cat cat1;
    cat1.sound();       // se afiseaza meow

    dog dog1;
    dog1.sound();       // se afiseaza wof

    cat* catp=new cat;
    catp->sound();      // se afiseaza meow

    dog* dogp=new dog;
    dogp->sound();      // se afiseaza wof

    // Putem sa facem un animal* pointer catre un obiect de tip cat sau dog.
    // Putem, pentru ca fiecare cat sau dog are o parte de animal.

    animal* animal1 = new cat;
    animal* animal2 = new dog;

    animal1->sound();   // acum se afiaseaza meow
    animal2->sound();   // acum se afiseaza wof

    return 0;
}
```

Cum functioneaza acest virtual si de ce ne-a rezolvat problema? Pe aceleasi 3 clase, vom modifica putin codul:

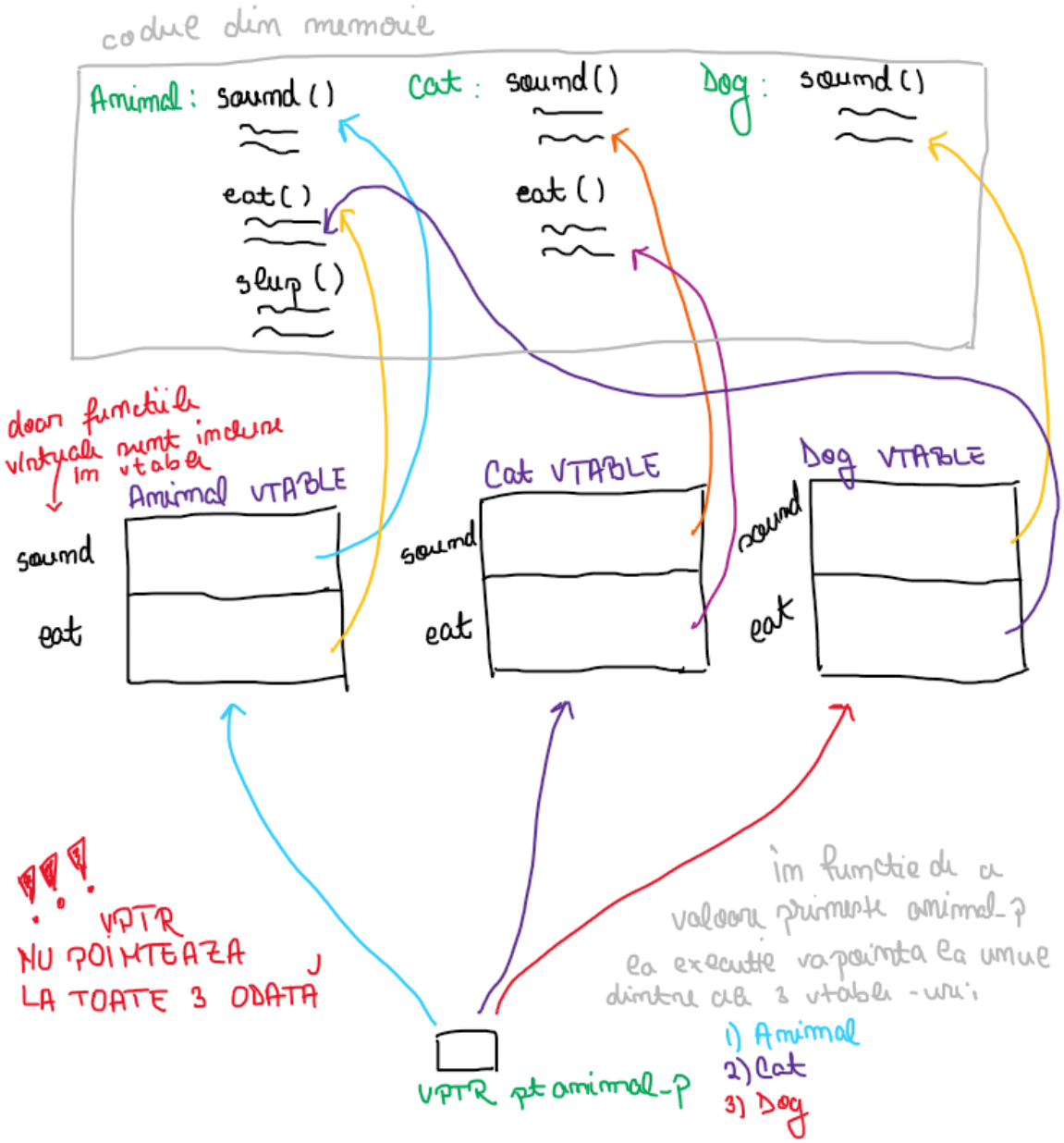
```
#include <iostream>
#include <ctime>
using namespace std;

class animal
{
public:
```

```
    virtual void sound() {cout<<"undefined \n";}
    virtual void eat() {cout<<"undefined\n";}
    void sleep() {};
};

class cat : public animal
{
public:
    void sound() {cout<<"meow \n";}
    void eat() {cout<<"mouse \n";}
};

class dog : public animal
{
public:
    void sound() {cout<<"wof \n";}

};


int main()
{
    srand(time(0));             // Facem seed pentru random.
    animal* animal_p;
    if(rand()%2==1)             // rand() ne da un numar random.
    {                           // Noi nu avem de unde sa stim valoarea
        animal_p = new dog;     // sa la compilare.
    }                           // Deci avem nevoie de un mecanism care
                                // sa stabileasca ce functie se apeleaza
                                // cand apelam animal_p, dar la run time.
                                // Ca la compile time nu avem informatii
                                // despre catre ce pointeaza animal_p.
                                // Acest mecanism se numeste late binding.
    else
    {
        animal_p = new cat;
    }
    animal_p->sound();
    return 0;
}
```

Late binding = un mecanism rpin care compilatorul adauga cod care identifica tipul de obiect la run time si apoi potriveste functiile la apelul de functie.

Cum se fac toate lucrurile astea?

- vtable: un tabel de pointeri la functii, mentinut pentru fiecare clasa.
- vptr: un pointer la vtable, mentinut pentru fiecare obiect



Constrangeri pentru suparincarcarea functiilor virtuale:

- functia initiala si supraincarcarea ai trebuie sa iaba aceelasi return type.

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

class animal
{
public:
    virtual void sound() {cout<<"undefined \n";}
};

class cat : public animal
{
```

```
public:
    int sound()             // eroare: int cat::sound()
                            // return type is not identical to nor
                            // covariant with return type "void" of
                            // overridden virtual function "animal::sound"
    {
        cout<<"meow \n";
        return 0;
    }
    void eat() {cout<<"mouse \n";}
};

int main()
{
    animal* animal_p = new cat;
    animal_p->sound();
    return 0;
}
```

- Nu apelati functii virtuale din constructori sau destructori. Stim ca intr-un obeict de tip derivat, mai intai se creeaza baza. Daca apelam o functie virtuala din constructorul bazei, partea derivata nici nu a fost creata inca. Se va apela automat implementarea din baza a functiei virtuale.

  La fel pentru destructori, mereu se va apela implementarea din baza.

De ce nu am face toate functiile virtuale? Pentru ca e ineficient. Vom face doar pentru functiile pentru care avem nevoie.

2.2.1 Functii pur virtuale si clasele abstracte

Functiile pur virtuale: functiile care nu au o implementare.
Mai devreme aveam:

```
class animal
{
```

```
public:
    virtual void sound() {cout<<"undefined \n";}
};
```

In loc sa facem acesta functie sa afiseze "undefined", puteam sa o facem pur virtuala:

```
class animal
{
public:
    virtual void sound()=0;
};
```

Acum clasa `animal` a devenit **abstracta**. Deci ce este o clasa abstracta? O clasa care contine o functie pur virtuala.

Cu ce sunt clasele abstracte mai speciale? Nu pot instantia obiecte dintr-o clasa abstracta.

In cazul acesta:

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

class animal                            // clasa abstracta
{
public:
    virtual void sound()=0;             // functie pur virtuala
};

class cat : public animal               // nu e clasa abstracta ca a implementat
                                        // functia pur virtuala mostenita
{
public:
    void sound()                        // implementarea functiei pur virtuale mostenite
    {
        cout<<"meow \n";
    }
    void eat() {cout<<"mouse \n";}
};
class dog : public animal               // e functie abstracta ca nu a implementat functia
                                        // pur virtuala pe care a mostenit-o din animal
{
public:
    void sleep() {cout<<"zzzzzzzzz \n";}
};

class orange_cat : public cat           // nu e clasa abstracta ca mosteneste implementarea
                                        // functiei pur virtuala de la cat
{
    void eat() {cout<<"fancy food\n";}
};

int main()
{
    animal ob1;                         // eroare: object of abstract class type "animal"
                                        // is not allowed: -- function "animal::sound" is
                                        // a pure virtual function
    cat ob2;
    dog ob3;                            // object of abstract class type "dog" is not
```

```
                                        // allowed: -- pure virtual function
                                        // "animal::sound" has no overrider
    orange_cat ob3;
    return 0;
}
```

2.2.2 Functii pur virtuale cu corp

Putem sa ii dam un cord functiei pur virtuale, dar ea ramane tot pur virtuala. Putem sa ne folosim de acel corp in derivate:

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

class animal
{
public:
    virtual void sound()=0;
};
void animal::sound()
{
    cout<<"nothingness";
}

class cat : public animal
{
public:
    void sound()
    {
        animal::sound();
    }
    void eat() {cout<<"mouse \n";}
};
int main()
{
    animal ob1;                 // tot eroare ca e clasa abstracta
    cat ob2;
    ob2.sound();                // afiseaza "nothingness"
    return 0;
}
```

3 Copy elision

Copy elision este o optimizare a compilatorului pentru a evita copierea unnecessary a obiectelor.
Exemplu:

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

class B
{
public:
    B(const char* str = "\0") //default constructor
    {
        cout << "Constructor called" << endl;
```

```
    }

    B(const B &b)  //copy constructor
    {
        cout << "Copy constructor called" << endl;
    }
};

int main()
{
    B ob = "copy me";  // se afiseaza "Constructor called"
    return 0;
}
```

De ce nu se apeleaza copy constructorul? Daca compilatorul foloseste copy elision, in loc sa transforme linia:

```
B ob = "copy me";
```

in

```
 B ob = B("copy me");
```

o transforma in:

```
B ob("copy me");
```

Daca aveti la exemen ceva de genul, puteti sa spuneti ce se intampla daca copy elision nu este activat si ce se intampla daca este activat.

4 Sizeof de o clasa

In C++ putem sa scriem pentru tipurile de date:

```
int main()
{
    cout<<sizeof(int)<<"\n";         // se afiseaza 4 pentru ca int-ul sta pe 4 bytes
    cout<<sizeof(char)<<"\n";        // se afiseaza 1
    cout<<sizeof(string)<<"\n";      // se afiseaza 24
    return 0;
}
```

Dar si clasele sunt tipuri de date, deci ce se intampla daca scriem:

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

class B
{};

int main()
{
    cout<<sizeof(B)<<"\n";       // se afiseaza 1
    return 0;
}
```

Dar daca am:

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

class B
{
    int x;      // ocupa 4 bytes
    char c;     // ocupa 1 byte
};

int main()
{
    cout<<sizeof(B)<<"\n";      // se afiseaza 8 , adica 4 + 1 rotunjit la
                                // un multiplu de max(4,1) = 4
    return 0;
}
```

Alt exemplu:

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

class B
{
    double x;       // ocupa 8 bytes
    char c[17];     // ocupa 1*17 bytes
};

int main()
{
    cout<<sizeof(B)<<"\n";      // se afiseaza 32 , adica 8 + 17 rotunjit
                                // la un multiplu de max(8,1) = 8
    return 0;
}
```

5 this

this este un pointer "ascuns", care pointeaza catre obiectul care a apelat metoda nestatica.

El este un exemplu de pointer constant catre o adresa neconstanta. Obiectul in sine se poate schimba, insa pointerul this nu poate pointa niciodata catre altceva decat obeictul care a apelat metoda.

6 Functii constante

Functiile constante sunt functiile care promit sa nu modifice obiectul care a apelat functia. Astfel, ca sa avem o functie constanta, trebuie ca functia sa nu fie statica (sa aiba pointerul this).

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

class B
{
public:
    void f () const
    {
        _x=4;           // error: assignment of member 'B::_x'
                        // in read-only object
```

```
    }
private:
    int _x;
};

int main()
{
    B ob1;
    ob1.f();
    return 0;
}
```

La ce ne ajuta functiile constante? Obiectele constante nu pot apela decat functii constante:

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

class B
{
public:
    void f ()
    {
        _x=4;
    }
private:
    int _x;
};

int main()
{
    const B ob1;
    ob1.f();            // the object has type qualifiers that
                        // are not compatible with the member
                        // function "B::f" -- object type is: const B
    return 0;
}
```

Don’t be like this, guys!