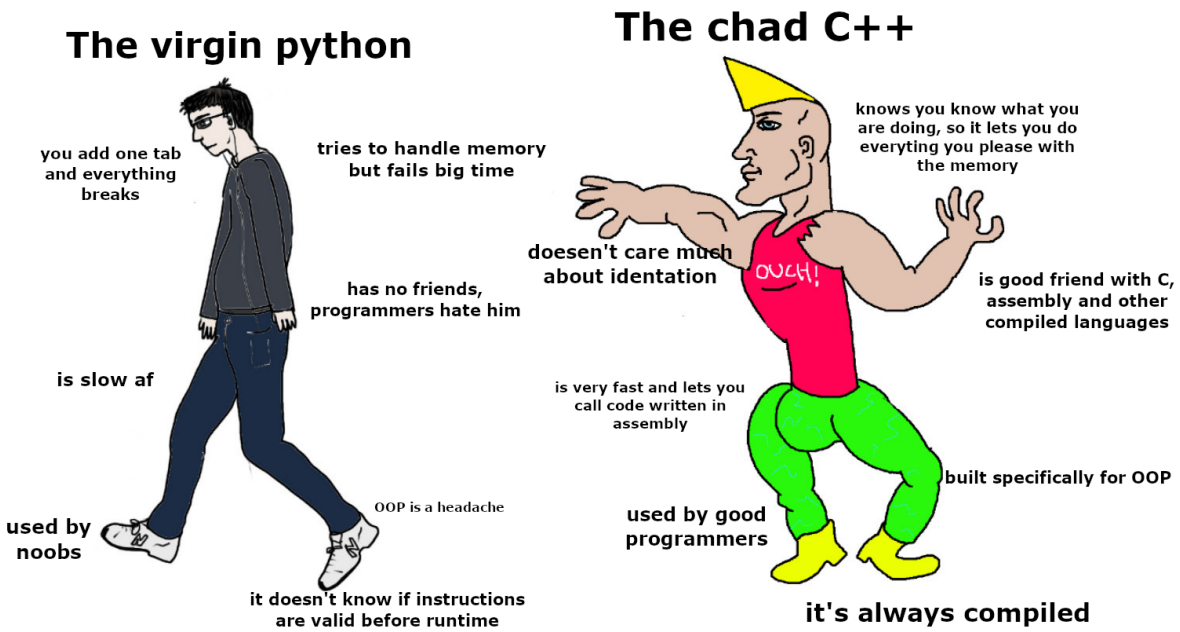
Tutoriat 5 POO

Bianca-Mihaela Stan, Silviu Stăncioiu

April 2021



1 Excepții

În C, pentru a verifica dacă o instrucțiune se execută cum trebuie, de regulă trebuie să punem un *if* după acea instrucțiune, care să verifice dacă instrucțiunea s-a executat cum trebuie. Exemplu:

```c
int* vec = malloc(sizeof(int) * n);       // incearca sa aloce dinamic
                                          // un array cu n elemente în C
if (!vec)                                 // în cazul în care vec este un pointer
                                          // null, atunci inseamna ca alocarea
                                          // a esuat
{
    printf("Could not alloc memory :("); // daca nu a reușit să aloce memorie
                                          // afiseaza pe ecran un mesaj și
    return -1;                            // returneaza -1.
}
```

În C++ există un mecanism care ne scutește de astfel de verificări după fiecare instrucțiune, acest mecanism se numește *Exception Handling*. Tot ce presupune acest sistem, este ca atunci când ceva nu se execută cum trebuie, să se arunce o excepție, iar programul să continue dintr-o zonă specială unde se va scrie cod care gestionează problema. Fie următorul exemplu:

```cpp
#include <iostream>

using namespace std;

int main()
{
    try                                                     // aici ii spunem sa
                                                            // incerce sa ruleze
                                                            // codul din
                                                            // interiorul acestui
                                                            // bloc de cod, iar
                                                            // daca nu reuseste,
                                                            // sa se opreasca
                                                            // si sa ruleze codul
                                                            // din blocul de catch
    {
        unsigned long long x = 10000000000000000;
        int* a = new int[x];                                // probabil pe niciun
                                                            // calculator nu merge
                                                            // sa alocam asa de
                                                            // multa memorie. Deci
                                                            // se va arunca
                                                            // o exceptie de catre
                                                            // calculator. Dupa
                                                            // ce se arunca
                                                            // exceptia, nu se mai
                                                            // continua executia
                                                            // in blocul try
                                                            // si se continua din
                                                            // blocul catch.
        delete[] a;                                         // la aceasta linie,
                                                            // probabil niciun
                                                            // calculator nu o sa
                                                            // ajunga
    }
    catch (exception e)                                     // aici scriem codul
                                                            // prin care gestionam
                                                            // problema. Note:
                                                            // daca nu apare nico
                                                            // problema, atunci
                                                            // pe acest cod
                                                            // nu se intra
    {
        cout << "My man, i got bad news for you" << endl;   // se vor afisa pe
        cout << "something went wrong.";                    // ecran aceste mesaje
                                                            // in cazul in care
                                                            // alocarea esueaza
    }

    return 0;
}
```

Avantajul acestui sistem, este că indiferent ce linie poate eșua din blocul de *try*, este nevoie doar de blocuri de catch pentru a gestiona problemele, nu mai este nevoie să scriem *if-uri* pentru a verifica fiecare linie din blocul *try*. Acum pur și simplu, la linia la care e problema se oprește, și intră în *catch*.

Observații:

- Dacă apelăm o funcție care aruncă o excepție în interiorul blocului de *try*, atunci aceasta va fi prinsă și se va executa codul din *catch*.
- Dacă apelăm o funcție care aruncă o excepție în interiorul blocului de *try*, dar acea excepție este prinsă într-un *try* din interiorul funcției, atunci *try-ul* din exterior nu va mai intra pe *catch-ul* lui, și se va executa în continuare de parcă nu s-a întâmplat nimic. Exemplu:

```
#include <iostream>

using namespace std;

void fun()
{
```

```
    try
    {
        unsigned long long x = 1000000000000000;
        int* a = new int[x];
        delete[] a;
    }
    catch (exception e)                                    // programul va intra
                                                           // pe acest catch
    {
        cout << "Am prins exceptia in functie." << endl;
    }
}

int main()
{
    try
    {
        fun();                                             // functia isi
                                                           // gestioneaza singura
                                                           // tratarea
                                                           // exceptiilor. Deci
                                                           // programul se executa
                                                           // normal.
    }
    catch (exception e)                                    // aici nu se va mai
                                                           // intra, deoarece
                                                           // exceptiile
                                                           // sunt gestionate in
                                                           // functie.
                                                           // Daca exceptia nu ar
                                                           // fi fost prinsa in
                                                           // functie, atunci
                                                           // s-ar fi apelat
                                                           // aceasta parte de
                                                           // cod.
    {
        cout << "Am prins exceptia in exteriorul functiei." << endl;
    }

    return 0;
}
```


Exceptie
catch-ul
din main
catch-ul
din functie
catch-ul
din main

- Dacă folosim excepții în programele noastre, performanța programelor poate scădea destul de mult (gândiți-vă cum s-ar traduce excepțiile în MIPS...).

Până acum am văzut doar cum tratăm excepții care pot apărea datorită limbajului C++. Totuși, putem ca atunci când nu ne convine ceva, să aruncăm propriile excepții. O excepție se aruncă folosind cuvântul *throw*, urmat de un obiect. Acest obiect poate fi orice obiect, iar în funcție de el se decide pe care *catch* se va intra (vom discuta despre asta mai încolo, dar ideea e că putem avea mai multe catch-ul care se execută în funcție de tipul obiectului aruncat cu *throw*). Exemplu:

```
#include <iostream>

using namespace std;

int main()
{
    try
    {
        throw 5;                                                    // aici ii spunem ca
                                                                    // avem o problema,
                                                                    // deci lanseaza o
                                                                    // exceptie. Obiectul
                                                                    // acestei exceptii
                                                                    // este un numar (5).
        cout << "Aceasta linie de cod nu se executa";               // aici nu se va ajunge
                                                                    // niciodata deoarece
                                                                    // se lanseaza o
                                                                    // exceptie inainte
                                                                    // de aceasta linie,
                                                                    // deci programul
                                                                    // continua in catch
    }
    catch (int val)                                                 // aici prindem
                                                                    // exceptia. Argumentul
                                                                    // catch-ului este int,
                                                                    // deoarece noi ne
                                                                    // asteptam sa prindem
                                                                    // un int
    {
        cout << "Valoarea aruncata este " << val << endl;           // afisam pe ecran
                                                                    // valoarea cu care
                                                                    // s-a dat throw,
                                                                    // adica 5.
    }

    return 0;
}
```

1.1 Excepțiile și moștenirea

Am văzut în primele exemple că noi în catch prindeam în catch obiecte de tip *exception*. Multe excepții din C++ moștenesc această clasă, iar de regulă, pentru a defini propriile tipuri de excepții trebuie să crepm clase care moștenesc clasa *exception*. Exemplu:

```
#include <iostream>

using namespace std;

class graphics_exception : public exception     // avem o exceptie custom ceata
                                                // de noi.
{
public:
    const char* what() const noexcept override // suprascriem functia care
                                                // returneaza mesajul exceptiei.
    {
        return "There is a problem with your graphics code.";
    }
};

int main()
{
    try
    {
        throw graphics_exception();             // aruncam o exceptie de tipul
```

```
                                                // clasei definite de noi
    }
    catch (const exception& e)                  // se observa ca avem o referinta
                                                // constanta ca argument la catch
                                                // ci nu un obiect de tipul
                                                // exception. Am facut astfel
                                                // deoarece nu vrem sa avem
                                                // object slicing. Daca aveam
                                                // object slicing, atunci pe
                                                // ecran se afisa
                                                // "std::exception", in loc sa se
                                                // afisese mesajul din exceptia
                                                // noastra custom
    {
        cout << e.what() << endl;
    }

    return 0;
}
```

Când aruncăm o excepție cu un obiect, se va intra pe primul catch valid. În cazul de mai sus, am avut un singur catch care a prins o referință la un obiect de tip exception. Acest catch este valid deoarece clasa *graphics_exception* moștenește clasa *exception*. Fie următorul exemplu:

```
#include <iostream>

using namespace std;

class base
{
};

class derived : public base  // avem o clasa derivata
{
};

int main()
{
    try
    {
        throw derived();     // dam throw la un obiect de tipul clasei derivate
    }
    catch (const base& b)    // vom intra pe acest catch, deoarece este primul
                             // catch pe care poate intra. Obiectul mosteneste
                             // clasa base, deci poate intra pe acest catch
    {
        cout << "Am prins baza." << endl;
    }
    catch (const derived& d) // aici nu intra deoarece intra pe catch-ul de mai
                             // sus
    {
        cout << "Am prins derivata." << endl;
    }
    catch(...)               // asa arata un catch default, pentru cazul in care
                             // nu exista un catch care sa prinda obiectul
```

```
                            // aruncat. Evident, nu se va intra pe acest caz.
    {
        cout << "Nu am prins nici una, nici alta." << endl;
    }

    return 0;
}
```

După cum se vede, programul intra pe catch-ul cu base, nu pe cel cu *derived*, deoarece este primul catch scris. Dacă ordinea catch-urilor care prind base, respectiv derived era intersată, atunci intra pe catch-ul cu derived:

```
try
{
    throw derived();
}
catch (const derived& d) // in cazul asta intra pe derived deoarece este primul
                         // catch valid pe care poate intra
{
    cout << "Am prins derivata." << endl;
}
catch (const base& b)
{
    cout << "Am prins baza." << endl;
}
catch(...)
{
    cout << "Nu am prins nici una, nici alta." << endl;
}
```

Observații:

- Este recomandat ca de diecare dată, să se scrie prima oară catch-urile pentru clasele derivate, iar apoi cele pentru clasele de bază. (În Java chiar nu compilează dacă nu se respectă această regulă.)
- Dacă se aruncă pointeri, se va intra pe cazul care corespunde cu tipul pointerului, indiferent dacă se lucrează cu clase polimorfice (în clasa de bază există virtaul). Exemplu:

```
#include <iostream>

using namespace std;

class base
{
public:

    virtual void f()
    {
    }

    virtual ~base()
    {
    }
};

class derived : public base
{
```

```
public:
    void f() override
    {
    }
};

int main()
{
    try
    {
        base* ptr = new derived(); // obiectul nostru este de tip derivat,
                                   // dar noi vom arunca un pointer de tipul
                                   // clasei de baza
        throw ptr;
    }
    catch (derived* d)             // nu va intra aici, desi obiectul este de
                                   // tipul d, deoarece am aruncat un pointer
                                   // de tipul bazei. Nu ia in calcul mostenirea
                                   // in cazul pointerilor.
    {
        cout << "Am prins derivata." << endl;
        delete d;
    }
    catch (base* b)                // Va intra aici, deoarece am aruncat
                                   // un pointer de tipul bazei.
    {
        cout << "Am prins baza." << endl;
        delete b;
    }
    catch(...)
    {
        cout << "Nu am prins nici una, nici alta." << endl;
    }


    return 0;
}
```

2 Smart pointers

Noi cand alocam pointeri, trebuie sa ne asiguram ca ii si dezalocam. Bineinteles, sunt multe cazuri in care fie uitam, fie nu e evident ca nu se dezaloca memoria, si atunci avem o problema. Daca am face un o clasa al carui job ar fi sa tina un pointer, iar cand iese din scope sa il dezaloce?

```
#include <iostream>

using namespace std;

class A
{
private:
    int _x;
};
```

```
class pointer_to_A
{
private:
    A* _a;
public:
    pointer_to_A(A* a = nullptr) : _a(a) {}
    ~pointer_to_A() { delete _a; }
    A& operator*() const { return *_a; }
    A* operator->() const { return _a; }
};

int main()
{
    pointer_to_A f(new A());                // se creeaza un pointer de tip
                                            // A si nu e nevoie sa il dezalocam,
                                            // se dezaloca singur la iesirea din
                                            // scope
    return 0;
}
```

This seems like a pretty good idea, right? Doar ca are o mica mare problema.

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

using namespace std;

class A
{
private:
    int _x;
};

class pointer_to_A
{
private:
    A* _a;
public:
    pointer_to_A(A* a = nullptr) : _a(a)
    {
        cout<<"Se aloca memorie.\n";
    }
    ~pointer_to_A()
    {
        cout<<"Se dezaloca memoria.\n";
        delete _a;
    }
    A& operator*() const { return *_a; }
    A* operator->() const { return _a; }
};

int main()
{
    pointer_to_A f(new A());
```

```
    pointer_to_A g(f);          // initializare prin copy constructor
    return 0;
}
```

Chestia asta afiseaza:
`Se aloca memorie.`
`Se dezaloca memoria.`
`Se dezaloca memoria.`
Okay, we get it, avem copy constructorul default, care ne face un shallow copy. Am putea sa facem un copy constructor care sa creeze un alt obiect, astfel incat cei 2 pointeri sa nu pointeze catre acelasi obiect. Dar asta e costisitor, pentru ca trebuie sa tot facem copii ale obiectelor.

Aici intervine conceptul de move semantics.

3 Move semantics

Move semantics inseamna ca o clasa transfera ownership-ul pentru un obeict catre un alt obiect, nu sa ii faca o copie.

Hai sa vedem un exemplu de move semantics:

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

using namespace std;

class A
{
private:
    int _x;
};

class pointer_to_A
{
private:
    A* _a;
public:
    pointer_to_A(A* a = nullptr) : _a(a)
    {
        cout<<"Se aloca memorie.\n";
    }
    ~pointer_to_A()
    {
        if(_a != nullptr)                   // Aici am pus asa doar ca sa vedeti
                                            // ca un singur obiect are ownership
                                            // asupra pointerului. Nu se intampla
                                            // nimic daca dam delete de un nullptr,
                                            // deci nu era nevoie de acest if.
        {
            cout<<"Se dezaloca memoria.\n";
            delete _a;
        }
    }
    A& operator*() const { return *_a; }
    A* operator->() const { return _a; }
```

```
    pointer_to_A(pointer_to_A& ob)      // Copy constructor care implementeaza
                                        // move semantics.
    {
        _a= ob._a;                      // Transfera ownership.
        ob._a = nullptr;                // Se asigura ca ii "confisca" wonership-ul
                                        // celuilalt obiect.
    }

    pointer_to_A& operator=(pointer_to_A& ob)
    {
        if(&ob == this)                  // Evitam copieri unnecessary daca ni
            return *this;               // se cere sa alocam acelasi obiect.

        delete _a;                      // Aici obiectul e deja creat si trebuie
                                        // sa dezalocam ce memorie avea el.

        _a = ob._a;                     // La fel ca la copy constructor.
        ob._a = nullptr;
        return *this;
    }
};

int main()
{
    pointer_to_A f(new A());
    pointer_to_A g(f);          // initializare prin copy constructor
    return 0;
}
```

3.1 R-value references

In C++11 s-au introdus un nou tip de referinte, numite r-value references. Referintele cu care eram noi familiari pana acum se mai numesc si l-value references.

Sa ne amintim ce insemnau:

- lvalue: valori/obiecte/chestii din C++ care au o adresa identificabila in menorie (ex: variabile, functii)
- rvalue: chestii din memorie care nu au adrese identificabile in memorie (ex: 5, 6, 9- 10*3, obiecte temporare)

Deci care e diferenta dintre l-value references si rvalue references?

L-value references (refererintele normale cu care suntem obisnuiti) pot fi initializate doar cu l-values.

R-value references pot fi initializate doar cu r-values.

```
#include <iostream>

using namespace std;


int main()
{
    int x = 5;
    int &lref = x;      // l-value reference: se poate initializa doar cu l-values
    int &&rref = 6;     // r-value reference
    return 0;
}
```

Avantajele pentru r-value references:

- putem extinde "durata de viata" a obietelor
- referintele r-value ne permit sa modificam r-values

```
#include <iostream>

class Fraction
{
private:
        int m_numerator;
        int m_denominator;

public:
        Fraction(int numerator = 0, int denominator = 1) :
                m_numerator{ numerator }, m_denominator{ denominator }
        {
        }

        friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
        {
                out << f1.m_numerator << '/' << f1.m_denominator;
                return out;
        }
};

int main()
{
        auto &&rref{ Fraction{ 3, 5 } };        // referinta r-value catre un obiect temporar

    std::cout << rref << '\n';

        return 0;
} // abia aici se distruge obiectul temporar pentru ca aici iese din
  // scope rref
```

Un alt exemplu:

```
#include <iostream>


int main()
{
    int &&rref{ 5 };            // aici se creeaza un obiect temporar cu valoare 5
    rref = 10;
    std::cout << rref << '\n';  // se afiseaza 10

    return 0;
}// aici se "distruge" obiectul temporar, fiindca iese din scope rref
```

3.1.1 R-value references si functiile

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

```
void fun(const int &lref) // argumentele de tip l-value vor alege aceasta functie
{
        cout << "l-value reference to const\n";
}

void fun(int &&rref) // argumentele de tip r-value vor alege aceasta functie
{
        cout << "r-value reference\n";
}

int main()
{
        int x{ 5 };
        fun(x); // x e l-value, deci de apeleaza functia cu l-value reference
        fun(5); // 5 e r-value, deci se apeleaza functie cu r-value reference

        return 0;
}
```

3.2 Move constructor si move assignment

Well, daca copy constructorul si copy assignment operator fac o copie a obiectului, move ocnstructor si move assignment operator schimba ownership-ul asupra unwi resurse.

Sa vedem un exemplu:

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

using namespace std;

class A
{
private:
    int _x;
public:
    A()
    {
        cout<<"A\n";
    }
    ~A()
    {
        cout<<"~A \n";
    }
};

class pointer_to_A
{
private:
    A* _a;
public:
    pointer_to_A(A* a = nullptr) : _a(a)
    {
        cout<<"pointer_to_A \n";
    }
```

```
~pointer_to_A()
{
    if(_a != nullptr)                   // Aici am pus asa doar ca sa vedeti
                                        // ca un singur obiect are ownership
                                        // asupra pointerului. Nu se intampla
                                        // nimic daca dam delete de un nullptr,
                                        // deci nu era nevoie de acest if.
    {
        delete _a;
    }
    cout<<"~pointer_to_A \n";
}
A& operator*() const { return *_a; }
A* operator->() const { return _a; }

pointer_to_A(const pointer_to_A& ob)    // Asta e copy constructorul normal
                                        // isi pastreaza functionalitatea de
                                        // a face o copie.
{
    _a= new A();
    _a = ob._a;
    cout<<"cc pointer_to_A \n";
}

pointer_to_A& operator=(const pointer_to_A& ob)     // La fel si asta, e copy
                                                    // assignment operator normal.
{
    if(&ob == this)
        return *this;

    delete _a;


    _a= new A();
    _a = ob._a;
    cout<<"pointer_to_A operator= \n";
    return *this;
}

pointer_to_A(pointer_to_A&& ob) : _a(ob._a)         // Asta e move constructor.
{
    ob._a = nullptr;
    cout<<"pointer_to_A move constructor \n";
}

pointer_to_A& operator=(pointer_to_A&& ob)          // Asta e move assignment operator.
{
    if(&ob == this)
        return *this;

    delete _a;

    _a = ob._a;
    ob._a = nullptr;
```

```
        cout<<"pointer_to_A move operator=\n";
        return *this;
    }

};

pointer_to_A generate_pointers()
{
    pointer_to_A object(new A);          // Aici se afiseaza A, apoi pointer_to_A.
    return object;                       // La return se afiseaza pointer_to_A move operator=
                                         // pentru ca se returneaza o valoare temporara.
}

int main()
{
    pointer_to_A object1;                // Aici se afiseaza pointer_to_A.
    object1 = generate_pointers();
    return 0;                            // Dupa return se afiseaza destructorii in
                                         // ordinea inversa constructorilor.
}
```

Asta afiseaza:

```
pointer_to_A
A
pointer_to_A
pointer_to_A move operator=
pointer_to_A
A
pointer_to_A
```

3.2.1 Cand se apeleaza move constructor si move assignment operator?

Ca sa se apeleaza trebuie sa se indeplineasca 2 conditii:

- sa fie definite
- argumentul sa fie un r-value

Compilatorul ne da un move constructor si un move assignment operator default, dar ei fac shallow copy, deci nu il folosim niciodata.

Soo:

- Cand argumentul e un l-value, nu modificam argumentul. Daca am: cel mai logic lucru de facut aici e sa copierea, pentru ca poate ne folosim de b mai incolo.
- In schimb cand obiectul e un r-value, clar nu ne folosim de el mai incolo, e ceva temporar. Deci putem sa nu mai facem copiere, ca oricum e expensive. Mutam owenrship-ul asupra resurselor care ne intereseaza (pointeri), ca sa nu avem ownership din 2 parti in acelasi timp (cum se intampla la shallow copy) si sa nu avem probleme ca dezalocam memoria de 2 ori.

3.3 std::move

Sa luam exemplul:

```cpp
#include <iostream>
#include <random>
#include <ctime>

using namespace std;

class A
{
private:
    int _x;
public:
    A()
    {
        cout<<"A\n";
    }
    ~A()
    {
        cout<<"~A \n";
    }
};

class pointer_to_A
{
private:
    A* _a;
public:
    pointer_to_A(A* a = nullptr) : _a(a)
    {
        cout<<"pointer_to_A \n";
    }
    ~pointer_to_A()
    {
        if(_a != nullptr)                   // Aici am pus asa doar ca sa vedeti
                                            // ca un singur obiect are ownership
                                            // asupra pointerului. Nu se intampla
                                            // nimic daca dam delete de un nullptr,
                                            // deci nu era nevoie de acest if.
        {
            delete _a;
        }
        cout<<"~pointer_to_A \n";
    }
    A& operator*() const { return *_a; }
    A* operator->() const { return _a; }

    pointer_to_A(const pointer_to_A& ob)    // Asta e copy constructorul normal
                                            // isi pastreaza functionalitatea de
                                            // a face o copie.
    {
        _a= new A();
        _a = ob._a;
        cout<<"cc pointer_to_A \n";
    }
```

```
    pointer_to_A& operator=(const pointer_to_A& ob)      // La fel si asta, e copy
                                                         // assignment operator normal.
    {
        if(&ob == this)
            return *this;

        delete _a;


        _a= new A();
        _a = ob._a;
        cout<<"pointer_to_A operator= \n";
        return *this;
    }

    pointer_to_A(pointer_to_A&& ob) : _a(ob._a)          // Asta e move constructor.
    {
        ob._a = nullptr;
        cout<<"pointer_to_A move constructor \n";
    }

    pointer_to_A& operator=(pointer_to_A&& ob)           // Asta e move assignment operator.
    {
        if(&ob == this)
            return *this;

        delete _a;

        _a = ob._a;
        ob._a = nullptr;

        cout<<"pointer_to_A move operator=\n";
        return *this;
    }

};

// Functie ineficienta care apeleaza copierea de prea multe ori.
void swap(pointer_to_A& a, pointer_to_A& b)
{
    pointer_to_A temp {a};      // se apeleaza copy constructorul
    a=b;                        // se apeleaza copy assignment
    b=temp;                     // se apeleaza copy assignment
}

void swap_eficient(pointer_to_A& a, pointer_to_A& b)
{
    pointer_to_A temp {std::move(a)};       // se apeleaza move constructor
    a = std::move(b);                       // apeleaza move assignment
    b = std::move(temp);                    // apeleaza move assignment
}

int main()
{
```

```
    return 0;
}
```

4 Smart pointers din STL

4.1 unique_ptr

Acesta este cel mai simplu tip de smart pointer. Se asigură că un pointer este folosit într-un singur loc în cod. Dacă încercăm să dăm acest obiect ca parametru către o funcție sau ceva ne vom trezi cu o eroare de compilare. Doar În blocul în care există acest pointer există obiectul. Când se încheie acel bloc, obiectul este șters automat, nu mai este nevoie să îi dăm delete.

Observație: Nu putem copia pointerul în alt pointer de tipul *unique_ptr*, dar îl putem muta în altă variabilă de acest tip dacă avem nevoie.

Exemplu:

```
#include <iostream>
#include <memory>

using namespace std;

class gamedev
{
public:

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

    const int get_salary() const { return _salary; }

    ~gamedev()                                          // desi noi nu dam
                                                        // delete nicaieri
                                                        // la obiectele
                                                        // create, unique_ptr
                                                        // se va ocupa singur
                                                        // sa stearga
                                                        // obiectele cand nu
                                                        // mai sunt folosite
    {
        cout << "A hero died." << endl;
    }

private:

    int _salary;
};

void fun(unique_ptr<gamedev> other)                     // aceasta functie
                                                        // este imposibil de
                                                        // apelat, deoarece
                                                        // nu putem trimite
```

```
                                                    // un unique_ptr
                                                    // in alta pare
                                                    // fata de locul in
                                                    // care a fost
                                                    // delcarat.
{
}

int main()
{
    unique_ptr<gamedev> john(new gamedev(10));      // putem crea un nou
                                                    // unique_ptr
                                                    // folosind new, ca
                                                    // la pointerii
                                                    // clasici.
                                                    // se observa ca
                                                    // am folosit <>
                                                    // pentru a da tipul
                                                    // pointerului.
                                                    // acest mecanism se
                                                    // numeste template,
                                                    // vom discuta
                                                    // despre asta in
                                                    // urmatoarele
                                                    // tutoriate. Acum
                                                    // trebuie retinut
                                                    // doar ca asa ii
                                                    // dam tipul de
                                                    // obiect pe care
                                                    // il tine
                                                    // pointerul.
    unique_ptr<gamedev> silviu = make_unique<gamedev>(20); // putem folosi si
                                                    // functia
                                                    // make_unique (a
                                                    // fost introdusa
                                                    // recent in C++)

    unique_ptr<gamedev> other;                      // avem un
                                                    // unique_ptr fara
                                                    // valoare

    /* other = silviu; */                           // nu avem voie sa
                                                    // atribuim unui
                                                    // unique_ptr
                                                    // valoarea
                                                    // altui unique_ptr.
                                                    // lina aceasta
                                                    // ne-ar cauza o
                                                    // eroare de
                                                    // compilare daca
                                                    // nu ar fi
                                                    // comentata.
    /* fun(silviu);    */                           // si aceasta linie
                                                    // ar fi cauzat
```

```
                                                    // o eroare de
                                                    // compilare
                                                    // daca nu ar fi
                                                    // fost comentata,
                                                    // deoarece nu putem
                                                    // trimite un
                                                    // unique_ptr la
                                                    // o functie (si
                                                    // pentru ca
                                                    // gamedevii nu
                                                    // au voie sa aiba
                                                    // parte de fun).

other = move(silviu);                               // putem, in schimb,
                                                    // sa schimbam
                                                    // obiectul care
                                                    // tine pointerul
                                                    // nostru folosind
                                                    // std::move. Acum
                                                    // avem pointerul
                                                    // nostru in other,
                                                    // iar silviu este
                                                    // un pointer
                                                    // invalid (always
                                                    // has been)

cout << john->get_salary() << endl;                 // observam
                                                    // ca putem folosi
                                                    // smart pointerii
                                                    // din STL ca pe
                                                    // cei normali, prin
                                                    // operatorul -> .
                                                    // Acesti pointeri
                                                    // au acest operator
                                                    // suprascris astfel
                                                    // incat sa se
                                                    // comporte cum s-ar
                                                    // comporta si un
                                                    // pointer normal.
                                                    // se afiseaza 10.

/* cout << silviu->get_salary() << endl; */         // Aceasta linie
                                                    // ne-ar genera
                                                    // un segmentation
                                                    // fault daca nu
                                                    // ar fi comentata,
                                                    // deoarece silviu
                                                    // este acum un
                                                    // pointer invalid.

cout << other->get_salary() << endl;                // va afisa 20,
                                                    // deoarece am
                                                    // mutat pointerul
                                                    // din silviu
```

```
                                                    // in other.

    return 0;                                       // la sfarsit se
                                                    // vor sterge
                                                    // automat obiectele
                                                    // datorita lui
                                                    // unique_ptr.
                                                    // Nu mai trebuie
                                                    // sa dam delete.
                                                    // Destructorii se
                                                    // apeleaza fix cum
                                                    // ne-am astepta
                                                    // si in cazul
                                                    // pointerilor
                                                    // normali.
}
```

4.2 shared_ptr

Acest tip de pointer este asemănător cu *unique_ptr*, dar acesta poate fi folosit și în mai multe locuri, nu doar unde a fost declarat. Acest tip de pointer păstrează numărul de locuri în care este folosit obiectul într-o variabilă numită *reference counter*. Dacă acel *counter* devine 0, atunci pointerul este șters (se dă *delete*).

Exemplu de folosire:

```
#include <iostream>
#include <memory>

using namespace std;

class gamedev
{
public:

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

    const int get_salary() const { return _salary; }

    ~gamedev()
    {
        cout << "A hero died." << endl;
    }

private:

    int _salary;
};

void fun(shared_ptr<gamedev> other)                          // de data asta
                                                             // putem avea
                                                             // o functie
                                                             // care ia ca
                                                             // argument
                                                             // un shared_ptr.
                                                             // reference count-ul
                                                             // se va incrementa
                                                             // cu 1. Cand se iese
                                                             // din functie, se
                                                             // va decrementa din
                                                             // nou cu 1.
{
    cout << "Ahaha, I didn't crash! " << other->get_salary() << endl;
}

int main()
{
    shared_ptr<gamedev> john(new gamedev(10));               // observam ca putem
                                                             // crea pointeri fix
                                                             // ca la unique_ptr
    shared_ptr<gamedev> silviu = make_shared<gamedev>(20); // functia se cheama
                                                             // make_shared in
                                                             // cazul asta

    shared_ptr<gamedev> other;
```

```
    other = silviu;                                 // avem voie sa facem
                                                    // aceasta atribuire.
                                                    // ambii pointeri voi
                                                    // duce catre acelasi
                                                    // obiect. In cazul
                                                    // asta reference
                                                    // counterul se va
                                                    // incrementa cu 1
                                                    // pentru ca sunt 2
                                                    // referinte catre
                                                    // acelasi obiect.
    fun(silviu);                                    // avem voie sa
                                                    // facem acest apel.

    other = move(silviu);                           // putem face si move
                                                    // ca la unique_ptr.
                                                    // in cazul asta se
                                                    // decrementeaza
                                                    // reference
                                                    // counterul pentru
                                                    // ca se pierde una
                                                    // din referinte.

    cout << john->get_salary() << endl;             // afiseaza 10
    /* cout << silviu->get_salary() << endl; */      // din nou, silviu
                                                    // este un pointer
                                                    // invalid
    cout << other->get_salary() << endl;            // va afisa 20

    return 0;                                       // reference counter-
                                                    // urile vor ajunge
                                                    // la 0 (se apeleaza
                                                    // destructorii pt
                                                    // shared_ptr, care
                                                    // duc reference
                                                    // counter-uluile la
                                                    // 0), deci se vor
                                                    // sterge obiectele
                                                    // si se vor apela
                                                    // destructorii.
}
```

Observație: Acest sistem de reference counting nu este perfect. Dacă avem două obiecte care au referințe prin *shared_ptr* unul către celălalt, atunci putem să ne trezim cu un memory leak. Fiecare obiect va avea un reference count egal cu 1, deși obiectele nu mai sunt folosite nicăieri. Din moment ce reference counter-ul nu va ajunge la 0, obiectele nu vor fi sterse. Exemplu:

```
#include <iostream>
#include <memory>

using namespace std;

class b_class;
```

```cpp
class a_class
{
public:
    shared_ptr<b_class> b_reference;                // in clasa a_class avem
                                                    // un shared_ptr catre un
                                                    // obiect de tipul b_class

    ~a_class()
    {
        cout << "Object A was destroyed" << endl;   // acest destructor nu va
                                                    // fi apelat
    }
};

class b_class
{
public:

    shared_ptr<a_class> a_reference;                // in clasa b_class avem
                                                    // un shared_ptr catre un
                                                    // obiect de tipul a_class

    ~b_class()
    {
        cout << "Object B was destroyed" << endl;   // nici acest destructor
                                                    // nu va fi apelat
    }
};


int main()
{
    shared_ptr<a_class> a = make_shared<a_class>(); // cream doua obiecte
    shared_ptr<b_class> b = make_shared<b_class>(); // si le pastram in
                                                    // pointeri de tipul
                                                    // shared_ptr

    a->b_reference = b;                             // atribuim referintele
    b->a_reference = a;                             // in obiecte

    return 0;                                       // niciun destructor nu va
                                                    // fi apelat, deci vom avea
                                                    // memory leaks. Reference
                                                    // counter-urile nu vor
                                                    // ajunge la 0, pentru ca
                                                    // desi se distrug
                                                    // referintele din main,
                                                    // raman cele dependente din
                                                    // clase.
}
```

USE SMART POINTERS THEY SAID....

YOU WON'T HAVE MEMORY LEAKS

THEY SAID...

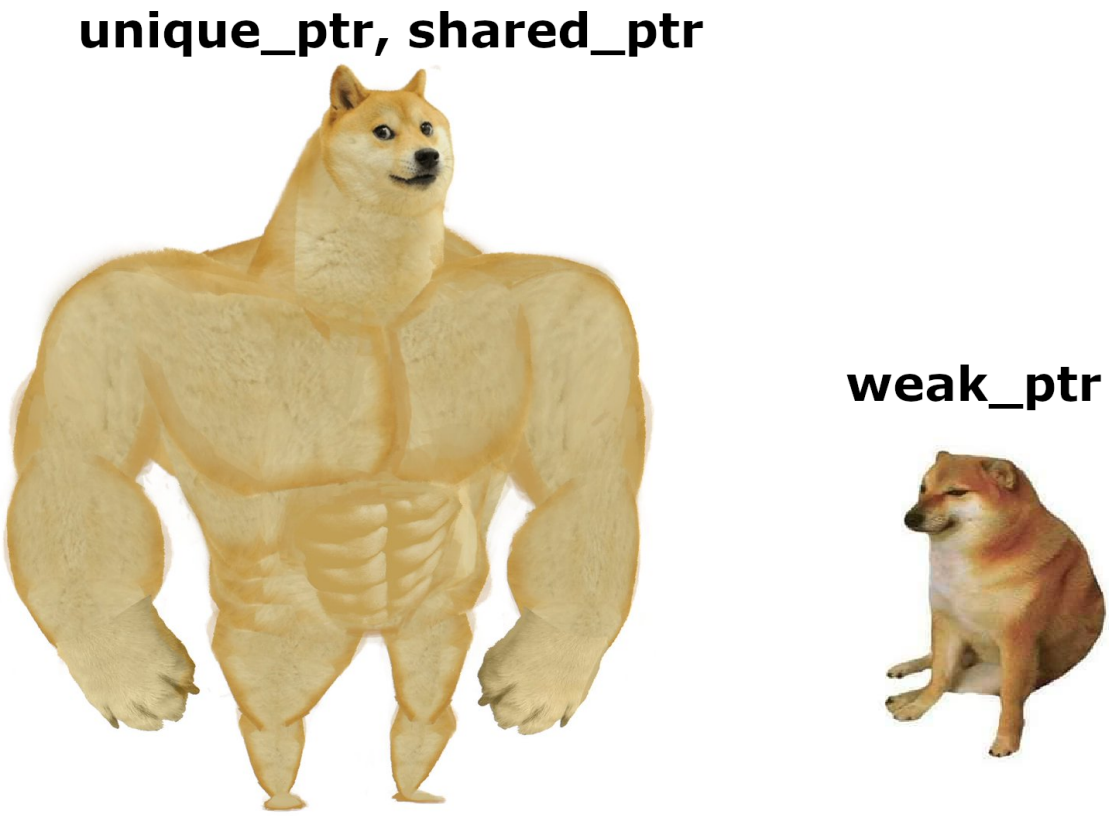
```
class a_class
{
public:
    shared_ptr<b_class> b_reference;
    ~a_class()
    {
        cout << "Object A was destroyed" << endl;
    }
};

class b_class
{
public:

    shared_ptr<a_class> a_reference;
    ~b_class()
    {
        cout << "Object B was destroyed" << endl;
    }
};
```

Bun, și nu există un workaround pentru chestia asta? You bet there it. Se cheamă *weak_ptr*, și despre asta vom vorbi în continuare.

4.3 weak_ptr



Acest tip de pointer a fost creat pentru a evita problema de mai sus. Este făcut să fie folosit împreună cu *shared_ptr*, dar acesta nu influențează cu nimic reference counter-ul. Asta înseamnă că putem avea cu *weak_ptr* și referințe care duc către obiecte care au fost deja șterse (dar e un risc pe care ni-l asumăm atunci când îl folosim, și încercăm să ne asigurăm că nu e cazul). Fie exemplul de mai sus, doar că acum vom folosi *weak_ptr*:

```
#include <iostream>
#include <memory>

using namespace std;

class b_class;

class a_class
{
public:
    weak_ptr<b_class> b_reference;              // acum aveam weak_ptr

    ~a_class()
    {
        cout << "Object A was destroyed" << endl;
```

```
    }
};

class b_class
{
public:

    shared_ptr<a_class> a_reference;              // aici am lasat un shared_ptr
                                                  // din motive didactice.

    ~b_class()
    {
        cout << "Object B was destroyed" << endl;
    }
};


int main()
{
    shared_ptr<a_class> a = make_shared<a_class>();
    shared_ptr<b_class> b = make_shared<b_class>();

    a->b_reference = b;                           // acum aceasta referinta este
                                                  // weak, deci nu se
                                                  // incrementeaza reference
                                                  // counter-ul pentru b
    b->a_reference = a;

    return 0;                                     // se vor apela destructorii.
                                                  // prima oara se va apela
                                                  // destructorul pentru B
                                                  // pentru ca acesta nu mai are
                                                  // nicio referinta. Acum ca B
                                                  // nu mai are nicio referinta,
                                                  // inseamna ca a fost distrusa
                                                  // si referinta shared catre A
                                                  // din clasa B. Deci acum nici
                                                  // obiectul A nu mai are nicio
                                                  // referinta (prima referinta
                                                  // a fost distrusa inainte
                                                  // de a se distruge cea de la
                                                  // B), asadar se va sterge
                                                  // si obiectul A, si se va
                                                  // apela destructorul sau.
                                                  // Am scapat de memory
                                                  // leak-uri. Daca ambele
                                                  // referinte din clase ar
                                                  // fi fost weak, atunci
                                                  // ordinea de apelare a
                                                  // destructorilor ar fi fost
                                                  // aceeasi (pentru ca
                                                  // obiectul B este mai nou
                                                  // decat A), dar in acest caz,
                                                  // ordinea de apel a
```

```
// destructorilor ar fi fost
// determinata doar de ordinea
// in care au fost declarati
// pointerii in main, nu si de
// referintele din interiorul
// obiectelor.
}
```

5 Exercitii

Exercitiul 1

```
int main()
{
        int x{};

        int &ref1{ x }; // A
        int &ref2{ 5 }; // B

        const int &ref3{ x }; // C
        const int &ref4{ 5 }; // D

        int &&ref5{ x }; // E
        int &&ref6{ 5 }; // F

        const int &&ref7{ x }; // G
        const int &&ref8{ 5 }; // H

        return 0;
}
```

Exercitiul 2

```
#include <iostream>
using namespace std;
int f(int y)
{
    if (y < 0)
        throw y;
    return y / 2;
}
int f(int y, int z)
{
    if (y < z)
        throw z - y;
    return y / 2;
}
float f(int& y)
{
    cout <<"y este referinta";
    return (float)y / 2;
}
int main()
{
```

```
    int x;
    try {
        cout <<"Da - mi un numar par : ";
        cin >> x;
        if (x % 2)
            x = f(x, 0);
        else
            x = f(x);
        cout <<"Numarul "<< x <<" e bun !"<< endl;
    }
    catch (int i) {
        cout <<"Numarul "<< i <<" nu e bun !"<< endl;
    }
    return 0;
}
```

Exercitiul 3

```
#include <iostream>
using namespace std;
class B
{ int i;
  public: B() { i=80; }
          virtual int get_i() { return i; }
};
class D: public B
{ int j;
  public: D() { j=27; }
          int get_j() {return j; }
};
int main()
{ D *p=new B;
  cout<<p->get_i();
  cout<<((D*)p)->get_j();
  return 0;
}
```

Exercitiul 4

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

```
    D(int i=0):B(i) {}
    operator B()
    {
        return B();
    };
    operator float()
    {
        return x;
    }
};
int main()
{
    D ob(12);
    try
    {
        throw ob;
    }
    catch (float a)
    {
        cout<<"A";
    }
    catch(B b)
    {
        cout<< "B";
    }
    catch(D d)
    {
        cout << "C";
    }
    catch(...)
    {
        cout<<"D";
    }
    return 0;
}
```

Exercitiul 5

```
#include <iostream>
using namespace std;
int f(int y)
{
    try {
        if (y > 0)
            throw y;
    }
    catch (int i) {
        throw;
    }
    return y - 2;
}
int f(int y, int z)
{
    try {
        if (y < z)
```

```
18              throw z - y;
19      }
20      catch (int i) {
21          throw;
22      }
23      return y + 2;
24  }
25  float f(float& y)
26  {
27      cout << " y este referinta";
28      return (float)y / 2;
29  }
30  int main()
31  {
32      int x;
33      try {
34          cout << "Da-mi un numar par: ";
35          cin >> x;
36          if (x % 2)
37              x = f(x, 0);
38          else
39              x = f(x);
40          cout << "Numarul " << x << " e bun!" << endl;
41      }
42      catch (int i) {
43          cout << "Numarul " << i << " nu e bun!" << endl;
44      }
45      return 0;
46  }
```

Exercitiul 6

```
#include <iostream>
using namespace std;
class A
{
    int x;
public:
    A(int i = 25)
    {
        x = i;
    }
```

```
    int& f() const
    {
        return x;
    }
};
int main()
{
    A ob(5);
    cout << ob.f();
    return 0;
}
```

Exercitiul 7

```
#include <iostream.h>
#include <typeinfo>
class B {
    int i;

public:
    B() { i = 1; }
    int get_i() { return i; }
};
class D : B {
    int j;

public:
    D() { j = 2; }
    int get_j() { return j; }
};
int main()
{
    B* p = new D;
    cout << p->get_i();
    if (typeid((B*)p).name() == "D*")
        cout << ((D*)p)->get_j();
    return 0;
}
```