

Keywords C++ - *static, default/delete, const*

Move Constructor

Tipuri de cast-uri în C++

Anul Universitar 2023 – 2024, Semestrul II

Programare Orientată pe Obiecte

LABORATOR 7, GRUPA 133

Cuprins

I. Cuvântul cheie <i>static</i> în diferite contexte.....	2
1. Funcții statice în clase	2
2. Date membre statice în clase	3
3. Variabile statice în funcții	4
4. Funcții/Variabile globale statice	4
II. <i>default</i> și <i>delete</i> explicații și exemple.....	5
1. <i>= default</i>	5
2. <i>= delete</i>	6
III. Constructor de mutare și <i>operator=</i> pentru atribuire prin mutare	7
1. Constructorul de mutare și operatorul de atribuire prin mutare	7
2. Legătura dintre constructorul de mutare și <i>=delete</i>	9
IV. Cuvântul cheie <i>const</i> în diferite contexte în C++	9
1. Utilizarea <i>const</i> în cadrul claselor	9
a) Utilizarea <i>const</i> pentru metode	9
b) Date membre <i>const</i> în clasă	9
V. Cast-uri în C++ - (<i>Tip</i>), <i>static_cast</i> , <i>const_cast</i> , <i>reinterpret_cast</i> , <i>dynamic_cast</i>	10
1. (<i>TipDate</i>) stil C	10
2. <i>static_cast</i> < <i>TipDate</i> >(<i>variabila</i>)	10
3. <i>const_cast</i> < <i>TipDate</i> >(<i>variabila</i>)	10
4. <i>reinterpret_cast</i> < <i>TipDate</i> >(<i>variabila</i>)	11
5. <i>dynamic_cast</i> < <i>TipDate</i> >(<i>variabila</i>)	11

Autor: Wagner Ștefan Daniel

I. Cuvântul cheie *static* în diferite contexte

Cuvântul cheie *static* poate fi utilizat în diverse contexte în C++, fiecare având implicații diferite asupra stocării și vizibilității. Aveți mai jos ilustrate sensurile semantice ale cuvântului cheie *static* în toate contextele:

1. Funcții statice în clase

Funcțiile statice din cadrul unei clase permit utilizatorului să le apeleze fără a avea un obiect de tipul acelei clase. Funcțiile statice pot acționa numai asupra datelor membre statice și asupra parametrilor primiți.

Dacă avem un obiect de tipul unei clase cu o metodă statică, putem să apelăm metoda din interiorul obiectului. Dar, de ce am face asta, dacă datele statice sunt ale clasei și nu ale obiectului? Nu prea are sens..

De asemenea, putem avea metode membre non-statice care acționează pe date statice.

Important: metodele statice nu conțin parametrul implicit *this*, deoarece pot fi apelate fără a avea nevoie de un obiect de tipul clasei.

Exemplu (ex01.cpp):

```
#include <iostream>

class Exemplu {
public:
    // definitie metoda statica
    static void Mesaj()
    {
        std::cout << "Aceasta este o functie statica." << std::endl;
    }
};

int main()
{
    Exemplu ex;
    // apel metoda statica din interiorul obiectului - posibil dar anti-pattern
    ex.Mesaj();
    // apel metoda statica fara obiect
    Exemplu::Mesaj();
    return 0;
}
```

2. Date membre statice în clase

În C++, datele membre statice sunt comune tuturor instanțelor claselor; Mai corect spus, atributele statice sunt ale clasei, fără să existe vreo instanță/obiect.

În exemplul de mai jos, variabila *numarStudenti* este comună tuturor obiectelor de tip *Student* (mai corect spus, aparține clasei și nu vreunui obiect), și este de asemenea accesibilă fără existența unui obiect, precum este ilustrat în *main()*.

O variabilă statică trebuie inițializată explicit în afara clasei într-un singur *.cpp*, fie fișierul asociat clasei (ideal), alternativ în *main.cpp*, pentru claritate.

Exemplu de lucru cu funcții și date membre statice (ex02.cpp):

```
#include <iostream>

class Student {
private:
    // Variabila statica comuna tuturor instantelor
    static int numarStudenti;
public:

    static void AdaugaStudenti(int numarStudenti_)
    {
        numarStudenti += numarStudenti_;
    }
    static int GetNumarStudenti(void) { return numarStudenti; }
};

int Student::numarStudenti = 0;

int main()
{
    // cu obiect
    Student student;
    int nrStudenti = 10;
    student.AdaugaStudenti(nrStudenti); // posibil dar nerecomandat (antipattern)
    std::cout << student.GetNumarStudenti() << std::endl; // idem

    // fara obiect
    Student::AdaugaStudenti(nrStudenti);
    std::cout << Student::GetNumarStudenti() << std::endl;

    return 0;
}
```

3. Variabile statice în funcții

În limbajele C/C++, variabilele statice declarate în cadrul funcțiilor se comportă precum variabilele globale (de fapt, ambele se află în aceeași zonă de memorie, segmentul de date, separat de stack și heap), însă scopul lor de vizibilitate este doar în cadrul acelei funcții.

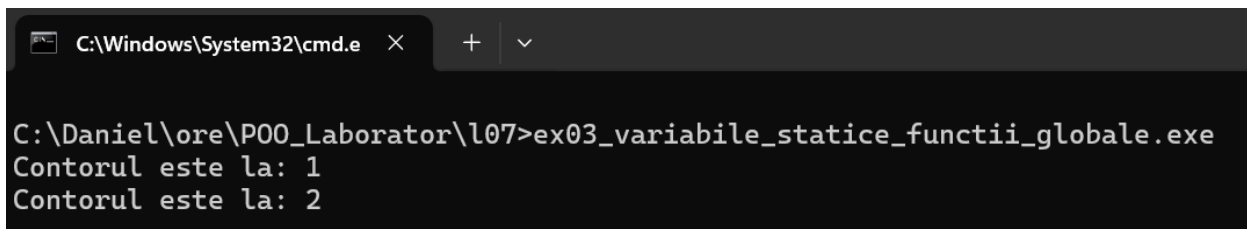
Altfel zis, își mențin valoarea pe parcursul apelului funcțiilor, însă nu sunt vizibile în afara funcției în care au fost definite.

Exemplu variabile statice în funcții (ex03.cpp):

```
#include <iostream>

void contor()
{
    // Valoarea lui cnt persista între apeluri
    // cnt este initializat doar la primul apel
    // precum o variabila globala, dar vizibila doar
    // in interiorul unei functii.
    static int cnt = 0;
    cnt++;
    std::cout << "Contorul este la: " << cnt << std::endl;
}

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



```
C:\Windows\System32\cmd.e  X  +  v

C:\Daniel\ore\P00_Laborator\l07>ex03_variabile_statice_functii_globale.exe
Contorul este la: 1
Contorul este la: 2
```

4. Funcții/Variabile globale statice

Funcțiile și variabilele declarate cu cuvântul cheie **static** în scopul global sunt vizibile doar în cadrul fișierului **.cpp** curent / assembly-ului curent generat. Dacă sunteți curioși, mai multe informații (deși nu excelente) găsiți aici: <https://www.geeksforgeeks.org/what-are-static-functions-in-c/>

II. *default* și *delete* explicații și exemple

În C++, *default* și *delete* sunt cuvinte cheie ale limbajului care anunță intenția de a folosi implementarea implicită (default), respectiv de a anula/interzice existența unei metode. Ambele sunt ilustrate mai jos în detaliu cu exemple:

1. *= default*

Poate fi folosit pentru a spune explicit compilatorului să genereze implementarea implicită pentru constructori, destructori, constructor de copiere, operator de atribuire.

Utilizarea lui poate îmbunătăți claritatea codului prin evidențierea intenției programatorului de a folosi comportamentul standard (**default**).

Observație: constructorul fără parametri/cc/op=/destructorul au implementarea *default* implicită, și acest lucru este doar pentru claritate.

Exemplu utilizarea cuvântului cheie *default* (ex04.cpp):

```
#include <iostream>

class Exemplu {
public:
    Exemplu() = default; // Constructor implicit
    Exemplu(const Exemplu&) = default; // Constructor de copiere
    Exemplu& operator=(const Exemplu&) = default; // Operator de atribuire
    ~Exemplu() = default; // Destructor
};

int main()
{
    // apelarea constructorului default fara parametri explicit
    Exemplu exemplu1{};
    // apelare constructor de copiere default explicit
    Exemplu exemplu2{ exemplu1 };
    // apelare operator= default
    exemplu1 = exemplu2;
    // ambii destructori default apelati implicit
    return 0;
}
```

2. = delete

Acest specificator este folosit pentru a “interzice” anumite funcții. Poate fi aplicat constructorilor, destructorilor, și operatorilor pentru a preveni utilizarea lor, fie implicită, fie explicită, de către compilator sau alte părți ale codului.

Exemplu utilizarea cuvântului cheie *delete* (ex05.cpp):

```
#include <iostream>

class FaraCopiere {
public:
    FaraCopiere() = default;
    FaraCopiere(const FaraCopiere&) = delete; // Interzice copierea
    FaraCopiere& operator=(const FaraCopiere&) = delete; // Interzice
    atribuirea prin copiere
};

int main()
{
    FaraCopiere instantia1;
    FaraCopiere instantia2;

    // function "FaraCopiere::FaraCopiere(const FaraCopiere &)" (declared at
    // line 6) cannot be referenced -- it is a deleted functionC/C++(1776)
    FaraCopiere instantia3{instantia1};

    // function "FaraCopiere::operator=(const FaraCopiere &)" (declared at
    // line 7) cannot be referenced -- it is a deleted functionC/C++(1776)
    instantia1 = instantia2;
    return 0;
}
```

Output compilare program cu erori la compile-time la apelul cc/op= care sunt funcții ”șterse”:

```
C:\Daniel\ore\P00_Laborator\l07>g++ -o ex05_delete ex05_delete.cpp
ex05_delete.cpp: In function 'int main()':
ex05_delete.cpp:17:36: error: use of deleted function 'FaraCopiere::FaraCopiere(const FaraCopiere&)'
    FaraCopiere instantia3{instantia1};
                                ^
ex05_delete.cpp:6:5: note: declared here
    FaraCopiere(const FaraCopiere&) = delete; // Interzice copierea
    ^~~~~~
ex05_delete.cpp:21:17: error: use of deleted function 'FaraCopiere& FaraCopiere::operator=(const FaraCopiere&)'
    instantia1 = instantia2;
                ^~~~~~
ex05_delete.cpp:7:18: note: declared here
    FaraCopiere& operator=(const FaraCopiere&) = delete; // Interzice atribuirea prin copiere
```

III. Constructor de mutare și *operator=* pentru atribuire prin mutare

1. Constructorul de mutare și operatorul de atribuire prin mutare

Constructorul de mutare în C++ permite transferul eficient al resurselor de la un obiect la altul. Este util atunci când avem obiecte temporare sau când doriți să transferați proprietatea ("ownership") asupra datelor fără costul suplimentar al copierii.

Același lucru este valabil și pentru operatorul de atribuire prin mutare, doar că, în **cazul *operator=*** la mutare, datele din obiectul destinație trebuie eliberate, pentru a evita memory leaks.

Exemplu constructor de mutare și locuri în care este apelat (ex06.cpp):

```
#include <iostream>

class Mutabil {
private:
    int *data;
    int count;
public:
    // Constructor standard
    Mutabil(int count_ = 0) : count{ count_ } {
        data = (count == 0) ? nullptr : new int[count];
        // initializare data array, ca sa aiba sens
    }
    // Constructor de mutare
    Mutabil(Mutabil&& sursa) noexcept : data{sursa.data}, count{sursa.count}
    {
        // am transferat ownership-ul datelor, obiectul curent devine gol
        sursa.data = nullptr;
        sursa.count = 0;
    }
    // operator de atribuire prin mutare
    Mutabil& operator=(Mutabil&& sursa) noexcept
    {
        if(this == &sursa)
            return *this;
        // toate datele alocate dinamic existente ale obiectului in care mutam
        // trebuiesc eliberate explicit, altfel avem memory leaks
        if(data != nullptr)
            delete[] data;

        // preiau prin shallow copy datele de la obiectul sursa
        data = sursa.data;
        count = sursa.count;
    }
};
```

```

        // golesc obiectul sursa fara sa eliberez memoria
        sursa.data = nullptr;
        sursa.count = 0;
        return *this;
    }
    // Utilizarea =delete pentru a bloca copierea si reatribuirea implicite
    Mutabil(const Mutabil&) = delete; // Blocheaza copierea
    Mutabil& operator=(const Mutabil&) = delete; // Blocheaza atribuirea
    ~Mutabil() { delete[] data; } // Destructor
};

Mutabil CreeazaMutabil(int numElems)
{
    return Mutabil(numElems);
}

void ProceseazaMutabil(Mutabil&& mutabil)
{
    // cod care se ocupa cu un obiect temporar
}

int main()
{
    int numElems = 1000;
    // initializez o instanta cu date de dimensiune relativ mare
    Mutabil instanta1 = Mutabil(numElems);
    // datele din instanta1 sunt mutate in instanta2, instanta1 ramane "goala"
    Mutabil instanta2 = std::move(instanta1);
    // datele din instanta2 au fost mutate in instanta 1 prin op= pentru mutare
    instanta1 = std::move(instanta2);
    // constructor de mutare invocat explicit deoarece avem un rvalue
    Mutabil instanta3 = Mutabil(numElems);
    // obiect temporar returnat prin intermediul constructorului de mutare
    Mutabil instanta4 = CreeazaMutabil(numElems);
    // constructor de mutare apelat pentru swap pentru eficienta
    std::swap(instanta1, instanta3);
    // am creat un obiect temporar rvalue, si il trimit ca parametru la functie
    ProceseazaMutabil(Mutabil(numElems));
    return 0;
}

```

Constructorul de mutare este deosebit de util pentru gestionarea resurselor în cazul obiectelor temporare, reducând overhead-ul asociat cu copierea datelor. În *main()*, sunt ilustrate mai multe modalități de a apela constructorul de mutare sau operatorul de atribuire pentru mutare, cu sens.

Important: *TipDate&&* este folosit pentru referințe către obiecte **rvalue**, temporare, care nu au adresă.

2. Legătura dintre constructorul de mutare și `=delete`

Atunci când definim un constructor de mutare, poate fi util să marcăm explicit constructorul de copiere și operatorul de atribuire prin copiere ca `=delete` pentru a preveni utilizarea lor accidentală. Acest lucru asigură că obiectele clasei sunt mutate eficient, fără a se recurge la copierea ineficientă a resurselor.

IV. Cuvântul cheie `const` în diferite contexte în C++

1. Utilizarea `const` în cadrul claselor

Cuvântul cheie `const` poate fi utilizat în cadrul claselor în C++ în următoarele moduri (dar și ca calificier pentru valoare de return, neilustrat aici):

a) Utilizarea `const` pentru metode

După cum am văzut și în laboratoarele anterioare, când trimitem un parametru (în mod special o referință către un obiect, dar nu numai), pe care nu intenționăm să îl modificăm în cadrul funcției/metodei, marcăm acel parametru drept `const`. Acest lucru este important pentru claritatea codului și pentru prevenirea erorilor accidentale.

Metodele unei clase marcate cu `const` la sfârșitul semnăturii/antetului/definiției metodei indică faptul că acea metodă nu modifică starea obiectului `this`, doar o utilizează într-un anumit fel. Aceste metode pot fi apelate și pe obiecte constante.

Exemple cunoscute: constructorul de copiere pentru orice clasă primește drept unic prim parametru `const T&`. Alte exemple au fost ilustrate pe parcursul laboratoarelor.

b) Date membre `const` în clasă

Un membru declarat `const` într-o clasă trebuie inițializat la momentul creării obiectului (în cadrul listei de inițializare) și nu poate fi modificat ulterior. Acest lucru este util pentru a marca explicit anumite date membre ale instanțelor claselor care nu ar trebui să își schimbe valoarea pe parcursul vieții obiectului.

Exemplu:

```
class ExempluVariabilaConstanta {
private:
    const int constValue;
public:
    // Initializarea constValue poate fi facuta doar in lista de initializare
    a constructorului.
    ExempluVariabilaConstanta(int val) : constValue{ val } {}
};
```

V. Cast-uri în C++ - (*Tip*), *static_cast*, *const_cast*, *reinterpret_cast*, *dynamic_cast*

Fiecare dintre aceste cast-uri ilustrate are utilizări specifice și trebuie ales în funcție de necesitățile specifice ale programului și de considerațiile legate de siguranță și performanță.

1. (*TipDate*) stil C

Cast-ul în stil C este cel mai de bază tip de cast și poate fi folosit pentru a converti un tip de date în altul. Este folosit pentru a converti între tipuri de date primitive sau pentru a converti pointeri între diferite tipuri de obiecte. Nerecomandat de folosit în C++ în alte contexte, însă posibil.

Exemplu (*TipDate*) cast stil C (ex07.cpp):

```
void CastStilC(void) {
    double pi = 3.1415926;
    // Cast stil C pentru a converti un double in int
    int pi_int = (int) pi;
}
```

2. *static_cast<TipDate>(variabila)*

static_cast<TipDate>(variabila) - este utilizat pentru conversii între tipuri compatibile, cum ar fi între tipuri numerice sau pentru a converti pointeri/referințe între clase care au o relație de moștenire. Este mai sigur decât cast-ul în stil C deoarece efectuează verificări la compilare (compile time checks).

Exemplu *static_cast<TipDate>(variabila)* (ex07.cpp):

```
void StaticCastExemplu(void) {
    class Baza {};
    class Derivata : public Baza {};
    Baza* b = new Derivata();
    // Conversia unui pointer de la clasa de baza la clasa derivata
    // Daca esueaza conversia - eroare de compilare
    Derivata* d = static_cast<Derivata*>(b);
}
```

3. *const_cast<TipDate>(variabila)*

const_cast<TipDate>(variabila) - folosit pentru a adăuga sau a elimina calificier-ul *const* de la variabile. Este util atunci când doriți să modificați o variabilă care a fost inițial declarată ca fiind *const*.

Exemplu *const_cast<TipDate>(variabila)* (ex07.cpp):

```
void ConstCastExemplu(void) {
    const int val = 10;
    int* modificabil = const_cast<int*>(&val);
    // Modificam valoarea, chiar daca originalul a fost declarat const
    *modificabil = 20;
}
```

4. *reinterpret_cast<TipDate>(variabila)*

reinterpret_cast<TipDate>(variabila) este folosit pentru conversii de tipuri pointer sau referințe la orice alt tip de pointer sau referință. Este cea mai puțin sigură formă de cast și ar trebui utilizată cu precauție deoarece poate duce la **undefined behaviour**.

Exemplu *reinterpret_cast<TipDate>(variabila)* (ex07.cpp):

```
void ReinterpretCastExemplu(void) {  
    // pe Windows 64bit, long long are 8bytes, dimensiunea unui pointer  
    long long ptr = 5323;  
    // Conversia unui long intr-un pointer char* (in cazul acesta, adresa  
    // invalida)  
    char* charPtr = reinterpret_cast<char*>(ptr);  
}
```

5. *dynamic_cast<TipDate>(variabila)*

dynamic_cast<TipDate>(variabila) este utilizat pentru conversii sigure la runtime între pointeri sau referințe într-o ierarhie de clasă. Această formă de cast este posibilă doar dacă există o relație de moștenire între tipurile implicate. Este folosit în principal pentru a determina tipul obiectului la runtime în cazul polimorfismului, și pentru validarea conversiei.

Spre deosebire de *static_cast<TipDate>(variabila)*, dacă *dynamic_cast<TipDate>(variabila)* eșuează asupra unui pointer, returnează **nullptr**. Dacă eșuează asupra unei referințe, aruncă excepția **std::bad_cast**.

Important: în ambele cazuri, nu mai este eroare de compilare, ci poate fi testată și gestionată direct în program

dynamic_cast este des utilizat pentru a determina dacă un obiect poate fi tratat în siguranță ca fiind de un anumit tip în ierarhia de clase. Acest lucru este extrem de util în cazul în care avem o referință sau un pointer către o clasă de bază, dar dorim să accesăm funcționalități specifice unei clase derivate, funcționalități pe care nu le putem accesa decât prin **downcasting** la clasa derivată corectă.

Avantaje

- **Siguranța Tipurilor:** Asigură că conversia între tipuri este validă, prevenind erori de comportament nedefinit (undefined behaviour).
- **Flexibilitate:** Permite gestionarea diferitelor tipuri de obiecte într-o manieră polimorfică.

Exemplu `dynamic_cast<TipDate>(variabila)` pentru downcasting (ex08.cpp):

```
#include <iostream>
#include <vector>

class Angajat {
public:
    virtual void AfiseazaRol() = 0;
    // virtual destructor daca clasele derivate alocă dinamic memorie
    // să fie apelat destructorul corect din clasa derivată, nu din bază
    virtual ~Angajat() {}
};

class Manager : public Angajat {
public:
    void AfiseazaRol() override {
        std::cout << "Manager\n";
    }
    void MetodaManager() {
        std::cout << "Metoda specifica manager\n";
    }
};

class Inginer : public Angajat {
public:
    void AfiseazaRol() override {
        std::cout << "Inginer\n";
    }
    void MetodaInginer() {
        std::cout << "Metoda specifica ingier\n";
    }
};

void ProceseazaAngajat(Angajat *angajat) {
    Manager *manager;
    Inginer *inginer;
    // metoda comuna poate fi apelată direct deoarece este virtuală
    angajat->AfiseazaRol();
    // metodele specifice obiectelor nu pot fi apelate decât după downcasting
    if ((manager = dynamic_cast<Manager*>(angajat)) != nullptr)
        manager->MetodaManager();
    else if ((inginer = dynamic_cast<Inginer*>(angajat)) != nullptr)
        inginer->MetodaInginer();
    else
        std::cout << "Tip necunoscut\n";
}
```

```

int main()
{
    std::vector<Angajat*> angajati;
    angajati.push_back(new Manager());
    angajati.push_back(new Inginer());

    for (Angajat *angajat : angajati)
    {
        ProcesoazaAngajat(angajat);
        delete angajat;
    }
    return 0;
}

```

Comentarii finale despre `dynamic_cast` și `ex08.cpp`

- În exemplul de mai sus, *`dynamic_cast`* este folosit pentru a determina tipul concret al obiectelor *`Angajat`*, permițând apelul metodelor specifice tipului real al obiectului derivat, precum *`MetodaInginer()`* pentru obiecte de tip *`Inginer`*, sau *`MetodaManager()`* pentru obiecte de tip *`Manager`*.
- Utilizarea *`dynamic_cast`* necesită ca cel puțin o clasă din ierarhie să aibă o metodă virtuală, asigurând astfel că clasele sunt polimorfice.