Supraîncărcarea Operatorilor

Anul Universitar 2023 – 2024, Semestrul II

Programare Orientată pe Obiecte

LABORATOR 3, GRUPA 133

Cuprins

Autor: Wagner Ștefan Daniel

I. Structura proiectului pentru clasa *Vector2*

1. Fișierul header *Vector2.h*

```
#ifndef VECTOR_2_H_
#define VECTOR_2_H_
#include <iostream>

class Vector2 {
private:
    float x;
    float y;
public:
    // Constructor cu parametri impliciti
    Vector2(float x = 0.0f, float y = 0.0f);
    // Getters
    float getX() const;
    float getY() const;
    // Setters
    void setX(float x);
    void setY(float y);

    // Supraincarcarea operatorului << pentru afisare
    friend std::ostream& operator<<(std::ostream& os, const Vector2& v);
    // Supraincarcarea operatorului >> pentru citire
    friend std::istream& operator>>(std::istream& is, Vector2& v);

    // In continuare: Exemple de supraincarcare a altor operatori cu sens
    // pentru clasa Vector2

    // Supraincarcarea operatorului + pentru adunarea a doi Vector2
    // si a salva rezultatul intr-un alt Vector2
    // Exemplu: Vector2 v = v1 + v2;
    Vector2 operator+(const Vector2& other) const;
    // Supraincarcarea operatorului += pentru a aduna un vector la
    // vectorul curent
    // Exemplu: v1 += v2;
    Vector2& operator+=(const Vector2& other);
    // Supraincarcarea operator- unar pentru a schimba semnul vectorului
    // Exemplu: Vector2 v = -v2;
    Vector2 operator-();

    // Alte definitii adaugate de voi din indrumar si din exercitii
};

#endif /* VECTOR_2_H_ */
```

2. Fișierul sursă *Vector2.cpp*

```cpp
#include "Vector2.h"

// Constructor cu parametri impliciti implementat cu liste de initializare
Vector2::Vector2(float x, float y) : x(x), y(y) {}

// Implementare Getters
float Vector2::getX() const { return x; }
float Vector2::getY() const { return y; }

// Implementare Setters
void Vector2::setX(float x) { this->x = x; }
void Vector2::setY(float y) { this->y = y; }

// Adaugati pe parcurs operatorii din indrumar

// Restul metodelor si operatorilor sugerati
```

3. Fișierul sursă *main.cpp*

```cpp
#include "Vector2.h"
// teoretic, nu trebuie sa includem iostream, ca Vector2.h il include dar
// include guards (#ifndef #define #endif), ne fac sa nu fie inclus multiplu
#include <iostream>
// fara using namespace std;

int main()
{
    // ilustrare functionalitati existente + implementate de voi
    return 0;
}
```

Observație importantă generală: ***const*** la finalul antetului metodei înseamnă că obiectul ***this*** este nemodificabil, deci l-am trimis ca prim parametru implicit constant. Are sens doar în cadrul claselor să folosim ***const*** la finalul antetului metodelor, deoarece funcțiile globale nu au parametrul ***this***. Folosiți ***const*** mereu când metodele nu modifică datele membre și doar le utilizează.

II. Introducere în supraîncărcarea operatorilor

În programarea orientată pe obiecte (POO), adesea este o practică uzuală să personalizăm comportamentul operatorilor standard (cum ar fi +, -, <<, >> etc.) pentru a lucra cu obiecte ale claselor noastre. Această personalizare se numește **supraîncărcarea operatorilor**.

În C++, operatorii sunt de fapt funcții, si supraîncărcarea operatorilor este de fapt "syntactic sugar", pentru a ușura lucrul cu obiectele.

Supraîncărcarea ne permite să definim modul în care operatorii standard funcționează cu instanțele claselor noastre, și la acest laborator veți vedea exemplificat și veți implementa câțiva dintre operatori pe un exemplu practic – clasa ***Vector2***.

1. Motivația supraîncărcării operatorilor în general

Scopul principal al supraîncărcării operatorilor este de a permite operații naturale și intuitive cu obiecte ale claselor noastre, similar cu tipurile de date fundamentale din C++. De exemplu, pentru clasa ***Vector2***, dorim să putem aduna doi vectori folosind operatorul + într-un mod care este direct și ușor de înțeles.

2. Ce operatori putem supraîncărca?

În C++, majoritatea operatorilor pot fi supraîncărcați, cu excepția unor operatori precum `::` (operatorul de rezoluție de scop), `.*` (operatorul de acces la membru prin pointer, specific C++), `.` (operatorul de acces la membrul unei structuri/uniuni/obiect) și `?:` (operatorul ternar), operatorul ***sizeof()***.

Putem supraîncărca următorii operatori:

1. operatorii aritmetici (+, -, *, /, %)
2. operatorii increment și decrement (++, --), pre si post fixați
3. operatorii de atribuire (=, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=)
4. operatorii de comparație (==, !=, <. >, <=, >=)
5. operatorii logici (!, &&, ||)
6. Operatorii pe biți (~, &, |, ^, <<, >>)
7. Operatorii [], (), ->, ., ?:, si operatorul virgulă, operatorul de cast la tip (sintaxă diferită).

Detalii suplimentare veți afla în cadrul cursurilor viitoare, dar și în referința oficiala aici.

III. Supraîncărcarea operatorilor de citire și afișare

1. Importanța supraîncărcării operatorilor << și >>

Operațiile de input/output sunt printre cele mai comune, și C++ ne oferă prin acești operatori speciali si tipurile de date ***std::istream*** si ***std::ostream*** șansa să afișăm cu ***std::cout*** și să citim cu ***std::cin*** orice obiect care implementează ***operator***<<, respectiv ***operator***>>. Rolul lor este esențial de asemenea la reutilizarea de cod, ceea ce veți vedea ulterior la moștenire, dar și la simpla compunere a claselor, spre exemplu, dacă toate obiectele implementează ***operator***<<, afișarea devine consistentă indiferent de tipul de date, primitiv sau creat de utilizator, deci pentru orice tip de date.

2. Supraîncărcarea *operator*<< (operatorul de afișare – *cout*)

Supraîncărcarea ***operator***<< pentru clasa ***Vector2*** ne permite să afișăm ușor obiectele ***Vector2*** folosind ***std::cout***. Prin aceasta, putem scrie cod care este clar și ușor de urmărit, fără a fi nevoie să apelăm explicit metode de afișare.

Vectorul este doar afișat și nu modificat, și deoarece nu putem face o citire prin copie și nu dorim lucru cu pointeri aici, **trimitem o referință constantă** către obiect:

```
// Supraincarcarea operatorului << pentru scriere Vector2
std::ostream& operator<<(std::ostream& os, const Vector2& v)
{
    os << '(' << v.x << ", " << v.y << ')';
    return os;
}
```

Caz general: Exemplul de mai sus poate fi generalizat pentru o clasă cu un obiect de orice tip:

```
// Supraincarcarea operatorului << pentru afisarea unui obiect oarecare
std::ostream& operator<<(std::ostream& os, const NumeClasa& obiect)
{
    // os << pentru toate datele membre relevante, cum ati fi folosit cout
    // este bine sa va obisnuiti sa formatati afisarea cat mai clar
    return os;
}
```

3. Supraîncărcarea *operator*>> (operatorul de citire – *cin*)

Similar, supraîncărcarea operatorului >> ne permite să citim ușor valorile pentru un obiect ***Vector2*** folosind ***std::cin***, facilitând interacțiunea cu utilizatorul într-un mod natural, fără necesitatea unei metode explicite de citire.

În cazul ***operator***>>, este de asemenea necesar să **trimitem prin referință** obiectul, însă vom modifica datele membre ale obiectul trimis, deci nu îl putem trimite ***const***.

```
// Supraincarcarea operatorului >> pentru citire Vector2
std::istream& operator>>(std::istream& is, Vector2& v)
{
    std::cout << "x: ";
    is >> v.x;
    std::cout << "y: ";
    is >> v.y;
    return is;
}
```

Caz general: Exemplul de mai sus poate fi generalizat similar precum ***operator***<<.

4. Observație importantă pentru supraîncărcarea operatorilor de citire și afișare și explicații funcții prietene (*friend*)

Pentru ***operator***<< și ***operator***>>: nu este necesar să mai specificăm ***friend*** deoarece doar în cadrul clasei anunțăm că o funcție este ***friend*** **(efect: va avea acces la variabilele *private*/*protected* din cadrul clasei)**, și nici să specificam ***Vector2::*** înainte de numele funcției, deoarece este funcție care nu este parte a clasei, ci funcție globală, stil C.

De regulă, pentru funcțiile din cadrul claselor preferăm să le numim metode (*methods*), însă funcțiile ***operator***<< și ***operator***>> sunt funcții prietene, deci nu aparțin clasei, drept concluzie este corect să le numim funcții și ar fi incorect să le numim metode.

De asemenea, fiind funcții prietene și nu parte a clasei, nu au parametrul implicit ***this***, deci trebuie să trimitem explicit obiectul în al doilea parametru.

IV. Alți operatori uzuali supraîncărcați

1. Supraîncărcarea *operator*+ pentru *Vector2*

```
Vector2 Vector2::operator+(const Vector2& other) const
{
    return Vector2(this->x + other.x, this->y + other.y);
}
```

2. Supraîncărcarea *operator*+= pentru *Vector2*

```
Vector2& Vector2::operator+=(const Vector2& other)
{
    this->x += other.x;
    this->y += other.y;
    return *this;
}
```

3. Supraîncărcarea *operator*- unar pentru *Vector2*

```
Vector2 Vector2::operator-()
{
    this->x = -this->x;
    this->y = -this->y;
    return *this;
}
```

Forma generală: Toți operatorii pe care limbajul C++ permite să fie supraîncărcați au o sintaxă similară:

```
TipDateReturnat NumeClasa::operator#([lista parametri])
{
  // Corpul functiei
}
```

TipDateReturnat poate fi tipul clasei (***operator+*** pe ***Vector2*** returnează tot ***Vector2***, un tip primitiv (***operator**** pentru produs scalar ***Vector2*** returnează ***float***), sau tipul altei clase.

[lista parametri] depinde de operatorul supraîncărcat:

- ***operator–*** și ***operator+*** unari, au lista de parametri vidă (exemplu capitolul **IV. 3.**)
- ***operator++*** și ***operator--*** prefixați au lista de parametri vidă, și postfixați primesc un parametru de tip ***int*** nefolosit (”*dummy*”)

Pentru restul operatorilor, cel mai bine citiți din referința oficială:
https://en.cppreference.com/w/cpp/language/operators

V. Constructor cu parametri impliciți și liste de inițializare

Declarația (antetul) constructorului cu parametri impliciți în ***Vector2.h***:

```
// Constructor cu parametri impliciti
Vector2(float x = 0.0f, float y = 0.0f);
```

Constructorul cu liste de inițializare (**Observație**: parameteri impliciți au fost adăugați doar în header, nu are sens să ii punem și in fișierul sursă) implementat în ***Vector2.cpp***:

```
// Constructor cu parametri impliciti implementat cu liste de initializare
Vector2::Vector2(float x, float y) : x(x), y(y) {}
```

Parametrii impliciți în constructori permit inițializarea obiectelor noastre cu valori prestabilite, oferind flexibilitate și facilitând crearea de instanțe fără a fi necesar să specificăm explicit toți parametrii. Acest lucru este util în multe contexte, cum ar fi atunci când vrem să creăm un vector cu coordonatele (0,0) fără a fi nevoie să transmitem aceste valori la fiecare construire a unui obiect ***Vector2***. Motivația din spatele acestei abordări este de a simplifica utilizarea clasei și de a oferi valori rezonabile prin 'default', reducând astfel riscul de eroare și volumul de cod necesar pentru inițializarea obiectelor, dar și de a reduce numărul constructorilor redundanți.

Listele de inițializare oferă o metodă eficientă și elegantă de a inițializa membrii unui obiect imediat în momentul construirii. Aceste liste permit asignarea valorilor membrilor direct, înainte de intrarea în corpul constructorului. Utilitatea lor este evidențiată în special în cazul inițializării membrilor constanți sau a referințelor, care trebuie inițializați în momentul declarației. Motivația pentru utilizarea listelor de inițializare este dublă: pe de o parte, îmbunătățesc performanța prin evitarea inițializărilor inutile, iar pe de altă parte, cresc claritatea codului, făcând inițializările mai vizibile și mai ușor de urmărit. În plus, în anumite cazuri, folosirea listelor de inițializare este obligatorie, cum ar fi pentru inițializarea membrilor care sunt obiecte ale unei clase ce nu are un constructor implicit sau pentru membrii ce sunt constanți sau referințe. Aceste situații necesită inițializarea explicită la momentul construirii obiectului, iar listele de inițializare oferă mecanismul necesar pentru a asigura acest proces în mod corect și eficient.

Adăugați următoarele antete de funcții în ***Vector2.h***, pentru a le implementa ulterior la exerciții:

```
Vector2 operator-(const Vector2& other) const;
Vector2& operator-=(const Vector2& other);
// produs scalar intre v1 si v2
float operator*(const Vector2& other) const;
// lungimea vectorului
// nu avem un operator echivalent deci implementam ca functie
float magnitude();
```

VI. Exerciții individuale

1) Pe baza codului de început cu ***Vector2.cpp***, ***Vector2.h*** si ***main.cpp***, realizați următoarele:

- **a)** Actualizați conținutul fișierului ***Vector2.cpp*** cu implementările metodelor definite în ***Vector2.h*** luate din în îndrumarul de laborator pe parcurs ce le înțelegeți. Testați fiecare funcționalitate în ***main.cpp***, imediat după ce ați implementat-o, astfel vă obișnuiți să fiți siguri că fiecare metodă adăugată este implementată corect. **Valabil în mod special la colocviu, să nu aveți surprize pe final!!!** Este bine să vă obișnuiți din timp sa faceți testare după implementare, vă va ajuta pe viitor chiar și în afara orelor de POO.
- **b)** Implementați și metodele al căror antet este definit in ***Vector2.h*** după propria voastră intuiție. Testați fiecare funcționalitate în ***main.cpp***.
- **c)** Cum ați putea implementa ***operator*==** și ***operator!=*** pentru clasa ***Vector2***? Dar **operator*** pentru înmulțirea cu un scalar? Scrieți antetele și implementați operatorii. Testați fiecare funcționalitate în ***main.cpp***.
- **d)** Ce alți operatori ar avea sens sa mai implementam pe ***Vector2***? Operatorii <, <=, >, >= au sens dacă comparăm magnitudinile vectorilor, deci ar fi operatori rezonabili de implementat. Scrieți antetele și implementați operatorii. Testați fiecare funcționalitate în ***main.cpp***.
- **e)** Observați faptul că este destul sa implementați ***operator*==*** și ***operator*<***, și implementați toți ceilalți operatori relaționali în funcție de cei doi existenți. Ce obținem? Reutilizare de cod, și dacă schimbăm implementarea, schimbăm doar în două locuri.

2) Similar clasei ***Vector2***, creați clasa ***Complex***, cu datele membre private ***double real*** și ***double imag***. Implementați operatorii care au sens, sesizând diferențele dintre cele două clase dar și similaritățile (pe lângă operatorii de mai sus, spre exemplu ***operator/*** care nu are sens la vectori, sau ***operator**** care are o semnificație diferită pentru numerele complexe. Momentan tratați împărțirea la zero ca o eroare si dați ***exit(EXIT_FAILURE);*** pentru care trebuie ***#include <cstdlib>***. Discutăm ulterior excepții. Implementarea voastră să fie împărțită în fișiere ***.h*** si ***.cpp***, cu un ***main.cpp*** de testare.

3) Creați un fișier copie a clasei ***List*** de la laboratorul 1, la care adaugați operator cu sens semantic astfel încât să puteți face ***append()***. Ilustrați în ***main()***.