

Laborator 03 - Notiuni avansate de C++

Responsabili

-  Andrei Vasiliu
-  Daniel Ciocîrlan

Obiective

În urma parcurgerii acestui laborator studentul va:

- afla funcționalitățile claselor/funțiilor prietene
- realiza supraîncărcarea operatorilor din C++
- înțelege conceptul de copy constructor
- înțelege conceptul de rule of three

Clase/metode prietene

Așa cum am văzut în primul laborator, fiecare membru al clasei poate avea 3 specificatori de acces:

- public
- private
- protected

Alegerea specificatorilor se face în special în funcție de ce funcționalitate vrem să exportăm din clasa respectivă.

Dacă vrem să accesăm datele private/protejate din afara clasei, avem următoarele opțiuni:

- Funcții care ne întorc/setează valorile membre
- Funcții/Clase prietene (friend) cu clasa curentă.

O funcție prieten are următoarele proprietăți:

- O funcție este considerată prietenă al unei clase, dacă în declararea clasei, este declarată funcția respectivă precedată de specificatorul **friend**
- Declararea unei funcții prieten poate fi făcută în orice parte a clasei (publică, privată sau protejată).
- Definiția funcției prieten se face global, în afara clasei.
- Funcția declarată ca **friend** are acces liber la orice membru din interiorul clasei.

O clasă prieten are următoarele proprietăți:

Search

- [Reguli generale și de notare](#)
- [Catalog](#)
- [Concursuri](#)
- [Calendar](#)

Articole

- [Laborator 1 - Introducere în C++](#)
- [Laborator 2 - Notiuni de C++](#)
- [Laborator 3 - Notiuni avansate de C++](#)
- [Laborator 4 - Stive](#)
- [Laborator 5 - Cozi](#)
- [Laborator 6 - Liste generice](#)
- [Laborator 7 - HashTable](#)
- [Laborator 8 - Grafuri](#)
- [Laborator 9 - Arbori Binari](#)
- [Laborator 10 - Arbori Binari de Căutare](#)
- [Laborator 11 - Heap-uri](#)
- [Laborator 12 - Treap-uri](#)

Laboratoare din anii trecuți

-  [Laboratoare 2012](#)
- [Laboratoare 2013](#)
- [Laboratoare 2014](#)

- O clasă B este considerată prieten al unei clase A, dacă în declararea clasei A s-a întâlnit expresia: `friend class B`
- Clasa B poate accesa orice membru din clasa A, fără nici o restricție.

De asemenea, dacă clasa A este considerată prieten cu clasa B, nu înseamnă că și clasa B este considerată prieten cu clasa A. Nici tranzitivitatea nu este valabilă în relația de prietenie dintre clase.

Exemplu:

```
class Complex{
private:
    int re;
    int im;
public:
    int GetRe();
    int GetIm();
    friend double ComplexModul(Complex c); //am declarat
    friend class Polinom; //Acum clasa Polinom care a
};

double ComplexModul(Complex c)
{
    return sqrt(c.re*c.re+c.im*c.im); //are voie, intru
}
```

Supraîncărcarea operatorilor

Un mecanism specific C++ este supraîncărcarea operatorilor, prin care programatorul poate asocia noi semnificații operatorilor deja existenți. De exemplu, dacă dorim ca două numere complexe să fie adunate, în C trebuie să scriem funcții specifice, nenaturale. În C++ putem scrie foarte ușor:

```
Complex a(2,3);
Complex b(4,5);
Complex c=a+b; //operatorul + a fost supraîncărcat pentru
```

Acest lucru este posibil, întrucât un operator este văzut ca o funcție, cu declarația:

```
tip_rezultat operator#(listă_argumente);
```

Așadar pentru a supraîncărca un operator pentru o anumită clasă, este necesar să declarăm funcția următoare în corpul acesteia:

```
tip_rezultat operator#(listă_argumente);
```

Există câteva restricții cu privire la supraîncărcare:

- Nu pot fi supraîncărcați operatorii: `::`, `.`, `*`, `?:`, `sizeof`.
- Setul de operatori ai limbajului C++ nu poate fi extins prin asocierea de semnificații noi unor caractere, care nu sunt

Laboratoare 2015

Teme

- Tema 1
- Tema 2
- Tema 3
- Tema 4
- Codificare Huffman
- Challenge

Resurse

- Debugging
- Data Structure Visualization

Table of Contents

- Laborator 03 - Notiuni avansate de C++
 - Obiective
 - Clase/metode prietene
 - Supraîncărcarea operatorilor
 - Operator supraîncărcat ca funcții prieten
 - Supraîncărcarea operatorilor << și >>
 - Operator supraîncărcat ca funcții membre
 - Supraîncărcarea operatorilor de atribuire
 - Copy-constructor
 - Când se apelează
 - Rule of Three

operatori, de exemplu nu putem defini operatorul `==` .

- Prin supraîncărcarea unui operator nu i se poate modifica aritatea (astfel operatorul `!` este unar și poate fi redefinit numai ca operator unar).
- Asociativitatea și precedența operatorului se mențin.
- La supraîncărcarea unui operator nu se pot specifica argumente cu valori implicite.

Operatori supraîncărcați ca funcții prieten

Un operator binar va fi reprezentat printr-o funcție nemembră cu două argumente, iar un operator unar, printr-o funcție nemembră cu un singur argument.

Utilizarea unui operator binar sub forma **`a#b`** este interpretată ca **`operator#(a,b)`**.

Argumentele sunt clase sau referințe constante la clase.

Supraîncărcarea operatorilor `<<` și `>>`

În C++, orice dispozitiv de I/O este văzut drept un stream, așadar operațiile de I/O sunt operații cu stream-uri, care se definesc în felul următor:

- **Citire:** se execută cu operatorul de extracție `»`, membru al clasei `istream`
- **Scriere:** se execută cu operatorul de inserție `«`, membru al clasei `ostream`

Acești operatori pot fi supraîncărcați pentru o clasă pentru a defini operații de I/O direct pe obiectele clasei.


Supraîncărcarea se poate efectua folosind funcții friend utilizând următoarea sintaxă:

```
istream& operator>> (istream& f, clasa & ob);  
ostream& operator<< (ostream& f, const clasa & ob);
```



Operatorii `»` și `«` întorc fluxul original, pentru a scrie în lăntuiri de tipul `f»ob1»ob2`.

Funcțiile operator pentru supraîncărcarea operatorilor de I/O le vom declara ca funcții prieten al clasei care interacționează cu fluxul.

 Complex.h

```
#include <iostream>  
  
class Complex  
{  
public:
```

```

double re;
double im;

Complex(double real=0, double imag=0): re(real),

//supraîncărcarea operatorilor +, - ca funcții de
friend Complex operator+(const Complex& s, const Complex& d)
friend Complex operator-(const Complex& s, const Complex& d)

//funcții operator pentru supraîncărcarea operatorilor
//declarate ca funcții de tip "friend"
friend std::ostream& operator<< (std::ostream& out, const Complex& z)
friend std::istream& operator>> (std::istream& is, Complex& z)
};

```

Complex.cpp

```

#include "complex.h"

Complex operator+(const Complex& s, const Complex& d)
{
    return Complex(s.re+d.re, s.im+d.im);
}

Complex operator-(const Complex& s, const Complex& d)
{
    return Complex(s.re-d.re, s.im-d.im);
}

std::ostream& operator<<(std::ostream& out, const Complex& z)
{
    out << "(" << z.re << ", " << z.im << ")" << std::endl;
    return out;
}

std::istream& operator>>(std::istream& is, Complex& z)
{
    is >> z.re >> z.im;
    return is;
}

```

main.cpp

```

#include "complex.h"

int main() {
    Complex a(1,1), b(-1,2);
    std::cout << "A: " << a << "B: " << b;
    std::cout << "A+B: " << (a+b);
    std::cin >> b;
    std::cout << "B: " << b;
    a=b;
    std::cout << "A: " << a << "B: " << b;
}

```

Funcțiilor membru li se transmite un argument implicit **this** (adresa obiectului curent), motiv pentru care un operator binar poate fi implementat printr-o funcție membru nestatică cu un singur argument.

Operatorii sunt interpretați în modul următor:

- Operatorul binar **a#b** este interpretat ca **a.operator#(b)**
- Operatorul unar prefixat **#a** este interpretat ca **a.operator#()**
- Operatorul unar postfixat **a#** este interpretat ca **a.operator#(int)**

Complex.h

```
#include <iostream>

class Complex
{
public:
    double re;
    double im;

    Complex(double real, double imag): re(real), im(imag) {}

    //operatori supraîncărcați ca funcții membre
    Complex operator+(const Complex& d);
    Complex operator-(const Complex& d);
    Complex& operator+=(const Complex& d);

    friend std::ostream& operator<< (std::ostream& out, const Complex& z) {
        out << "(" << z.re << "," << z.im << ")";
        return out;
    }
};
```

Complex.cpp

```
#include "complex.h"

Complex Complex::operator+(const Complex& d) {
    return Complex(re+d.re, im+d.im);
}

Complex Complex::operator-(const Complex& d) {
    return Complex(re-d.re, im-d.im);
}

Complex& Complex::operator+=(const Complex& d) {
    re+=d.re;
    im+=d.im;
    return *this;
}

std::ostream& operator<<(std::ostream& out, const Complex& z) {
    out << "(" << z.re << "," << z.im << ")";
    return out;
}
```

```

}

std::istream& operator>>(std::istream& is, Complex& z) {
    is >> z.re >> z.im;
    return is;
}

```

Supraîncărcarea operatorului de atribuire

Așa cum am amintit mai sus, majoritatea operatorilor pot fi supraîncărcați. O atenție importantă trebuie acordată operatorului de atribuire, dacă nu este supraîncărcat, realizează o copiere membru cu membru.

Pentru obiectele care nu conțin date alocate dinamic la inițializare, atribuirea prin copiere membru cu membru funcționează corect, motiv pentru care nu se supraîncarcă operatorul de atribuire.



Pentru clasele ce conțin date alocate dinamic, copierea membru cu membru, executată în mod implicit la atribuire conduce la copierea pointerilor la datele alocate dinamic, în loc de a copia datele.

Operatorul de atribuire poate fi redefinit numai ca funcție membră, el fiind legat de obiectul din stânga operatorului =, motiv pentru care va întoarce o referință la obiect.


 String.h

```

class String{
    char* s;
    int n; // lungimea sirului

public:
    String();
    String(const char* p);
    String(const String& r);
    ~String();
    String& operator=(const String& d);
    String& operator=(const char* p);
};

```

 String.cpp

```

#include "String.h"
#include <string.h>

String& String::operator=(const String& d) {

```

```

    if(this != &d){          //evitare autoatribuire
        if(s)                //curatire
            delete [] s;
        n=d.n;                //copiere
        s=new char[n+1];
        strcpy(s, d.s);
    }
    return *this;            //intoarce referinta la obiectul
}

String& String::operator=(const char* p){
    if(s)
        delete [] s;
    n=strlen(p);
    s=new char[n+1];
    strcpy(s, p);
    return *this;
}

```

Copy-constructor

Reprezintă un tip de constructor special care se folosește când se dorește/este necesară o copie a unui obiect existent. Dacă nu este declarat, se va genera unul default de către compilator.

Poate avea unul din următoarele prototipuri

- MyClass(const MyClass& obj);
- MyClass(MyClass& obj);

Când se apelează?

1) Apel explicit

 explicit_copy_constructor_call.cpp

```

MyClass m;
MyClass x = MyClass(m); /* apel explicit al copy-c

```

2) Transfer prin valoare ca argument într-o funcție

 call_by_value.cpp

```

void f(MyClass obj);
...
MyClass o;
f(o); /* se apelează copy-constructor */


```

3) Transfer prin valoare ca return al unei funcții

 return_by_value.cpp

```
MyClass f()
{
    MyClass a;
    return a; /* se apelează copy-constructor */
}
```

4) La inițializarea unei variabile declarate pe aceeași linie

 init.cpp

```
MyClass m;
MyClass x = m; /* se apelează copy-constructor */
```


Rule of Three

Reprezintă un concept de **must do** pentru C++. Astfel:



Dacă programatorul și-a declarat/definit unul dintre **destructor**, **operator de assignment** sau **copy-constructor**, trebuie să îi declare/definească și pe ceilalți 2

Explicație: dacă funcționalitatea vreunui dintre cei 3 se vrea mai specială decât cea oferită default, atunci mai mult ca sigur se dorește schimbarea funcționalității default și pentru ceilalți 2 rămași.

 rule_of_3.cpp

```
class Complex
{
private:
    int re;
    int im;
public:
    Complex(const Complex& c)
    {
        re = c.re;
        im = c.im;
        printf("copy constructor\n");
    }

    void operator=(const Complex& c)
    {
        re = c.re;
        im = c.im;
        printf("assignment operator\n");
    }

    ~Complex()
    {

```



```

        printf("destructor\n");
    }
};

```

1. [5p] Implementati clasa **Fractie**, cu următoarele particularități:
 - a. [2p] doi constructori:
 - I. primul vid
 - II. al doilea va primi ca argumente numitorul și numărătorul
 - b. Se vor implementa funcții membre pentru:
 - I. [0.5p] determinarea numitorului și numărătorului
 - II. [1p] supraîncărcarea operatorilor de comparație <, >, == (aveți grijă la egalitatea a două fracții)
 - III. [0.5p] supraîncărcarea operatorilor +, -
 - IV. [bonus 1p] Supraîncărcați operatorii ++ și -- unari astfel încât numărul rațional reprezentat să crească/scadă cu o unitate. Căutați să vedeți cum se face diferența între ++ prefixat și postfixat.
2. [5p] Implementați clasa template **Vector** care să permită lucrul cu vectori de obiecte, cu următoarele particularități:
 - a. Vor exista doi constructori:
 - I. primul, vid, va inițializa numărul de elemente la 0 și pointerul de elemente la NULL
 - II. al doilea va primi ca argument numărul de elemente și va alocă memorie pentru pointer
 - b. [0.5p] Se va defini și un destructor, care va dezaloca memoria alocată dinamic
 - c. Se vor implementa funcții friend (nemembre) pentru:
 - I. [1p] testul de egalitate a doi vector (supraîncărcarea operatorului ==)
 - II. [0.5p] supraîncărcarea operatorului « (pentru scriere)
 - III. [0.5p] supraîncărcarea operatorului » (pentru citire)
 - d. Se vor implementa funcții membre pentru:
 - I. [1p] supraîncărcarea operatorului de atribuire între două obiecte de tip vector
 - II. [0.5p] supraîncărcarea operatorului de indexare [] ce va permite accesul la elementele individuale prin indexare **Operatorul de indexare** este un operator binar, având ca prim termen obiectul care se indexează, iar ca al doilea termen indicele. *(obiect[indice] este interpretat ca obiect.operator[](indice).*




Tipul parametrului pentru copy-constructor trebuie să fie identic cu cel al parametrului pentru operatorul de assignment

1. [4p] Implementați clasa template **Set** care să permită lucrul cu mulțimi de obiecte, cu următoarele particularități:
 - a. Constructorul va primi dimensiunea maximă de elemente care pot fi ținute în mulțime și va alocă spațiul necesar.
 - b. [0.5p] Se va defini și un destructor, care va dezaloca memoria alocată dinamic.
 - c. Se vor implementa funcții membre pentru:
 - I. [1p] supraîncărcarea operatorului += pentru adăugarea unui nou element în mulțime (dacă elementul există deja în mulțime atunci nu va mai fi adăugat).
 - II. [0.5p] supraîncărcarea operatorului -= pentru eliminarea unui element din mulțime.
 - d. Se vor implementa funcții friend (nemembre) pentru:
 - I. [1p] testul de egalitate a două mulțimi (supraîncărcarea operatorului ==): două mulțimi sunt egale dacă conțin aceleași elemente.
 - II. [0.5p] supraîncărcarea operatorului « (pentru scriere).
 - III. [0.5p] supraîncărcarea operatorului » (pentru citire).

sd-ca/2015/laboratoare/laborator-03.txt · Last modified: 2017/03/04 22:20 by andrei.vasiliiu2211

 Old revisions

 Media Manager  Back to top

 BY-SA

CHIMERIC DE

WSC CSS

 DOKUWIKI

 GET FIREFOX

RSS XML FEED

WSC XHTML 1.0