Tutoriat 7 POO

Bianca-Mihaela Stan, Silviu Stăncioiu

April 2021

Exercitiile de POO be like

1 Template-uri

1.1 Clase specializate parțial

Să ne aducem aminte exemplul de la specializări parțiale din tutoriatul trecut:

```
#include <iostream>
#include <cstring>

using namespace std;

template<typename T, int size>
class basic_array
{
public:
    T* get_array()
    {
        return _arr;
    }

    T& operator[](int index)
    {
        return _arr[index];
```

```
    }

private:

    T _arr[size];
};

template<typename T, int size>
void print(basic_array<T, size>& arr)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

template<int size>
void print(basic_array<char, size>& arr)
{
    for (int i = 0; i < size; i++)
        cout << arr[i];

    cout << endl;
}

int main()
{
    basic_array<int, 3> test;

    for (int i = 0; i < 3; i++)
        test[i] = i;

    print(test);

    basic_array<char, 14> str14;
    strcpy(str14.get_array(), "Hello, world!");

    print(str14);

    basic_array<char,12> str12;
    strcpy(str12.get_array(), "Hello, man!");
    print(str12);

    return 0;
}
```

Ne aducem aminte că aveam o specializare parțială pentru funcția *print* care specializa funcția noastră doar pentru *char*, dar nu și pentru *size*. Este posibil că noi să vrem să specializăm o metodă a unei clase în felul acesta. Fie următoarea clasă template:

```
template<typename T, int size>
class basic_array
{
public:
    T* get_array()
    {
```

```
        return _arr;
    }

    T& operator[](int index)
    {
        return _arr[index];
    }

    void print()
    {
        for (int i = 0; i < size; i++)
            cout << _arr[i] << " ";
        cout << endl;
    }

private:

    T _arr[size];
};
```

Această clasă este fix clasa din exemplul de mai sus, doar că de data asta funcția print este o metodă a clasei, nu o funcție separată. Dacă am încerca să îi facem o specializare parțială acestei metode ca în cazul funcției din exemplul din tutoriatul trecut, atunci vom avea o eroare de compilare:

```
// nu putem avea specializare partiala pe o metoda a unei clase
template<int size>
void basic_array<char, size>::print()
{
    for (int i = 0; i < size; i++)
        cout << _arr[i];
    cout << endl;
}
```

Se observă că specializarea de mai sus dă o eroare de compilare, deci este nevoie să specializăm întreaga clasă pentru a putea specializa o metodă. Deci vom avea:

```
#include <iostream>
#include <cstring>

using namespace std;

template<typename T, int size>
class basic_array
{
public:
    T* get_array()
    {
        return _arr;
    }

    T& operator[](int index)
    {
        return _arr[index];
    }

    void print()
```

```
    {
        for (int i = 0; i < size; i++)
            cout << _arr[i] << " ";
        cout << endl;
    }

private:

    T _arr[size];
};

template<int size>
class basic_array<char, size> // avem o specializare partiala a clasei.
                              // se observa ca sintaxa este asemanatoare
                              // cu cea pentru specializari partiale
                              // la functii.
{
public:
    char* get_array()
    {
        return _arr;
    }

    char& operator[](int index)
    {
        return _arr[index];
    }

    void print()
    {
        for (int i = 0; i < size; i++)
            cout << _arr[i];
        cout << endl;
    }

private:

    char _arr[size];
};

int main()
{
    basic_array<int, 10> arr;

    for (int i = 0; i < 10; i++)
        arr[i] = i;

    arr.print();

    basic_array<char, 14> hlw;
    strcpy(hlw.get_array(), "Hello, World!");

    hlw.print();
```

```
    return 0;
}
```

Exemplul de mai sus merge, desi e un lucru care ne deranjează. Se observă că am cam dat cu copy-pasta la codul din clasa template și doar am înlocuit cu *char* T-ul din clasa template. Noi nu vrem să avem copy-pasta în cod, deci va trebui să găsim o soluție de a specializa clasa pentru a schimba funcția noastră de *print* în cazul *char*-urilor, dar să nu avem copy-pasta. MOȘTENIRE!!!

Nu noi nu vom mai specializa clasa noastră, ci în schimb vom transforma clasa noastră într-o clasă template de bază, iar apoi vom face alte clase template care moștenesc această clasă de bază, și pe care le vom specializa după bunul plac. Vom avea:

```
#include <iostream>
#include <cstring>

using namespace std;

template<typename T, int size>
class basic_array_base                                  // clasa de baza care va
                                                        // fi mostenita de
                                                        // clasele specializate

{
public:
    T* get_array()
    {
        return _arr;
    }

    T& operator[](int index)
    {
        return _arr[index];
    }

    void print()
    {
        for (int i = 0; i < size; i++)
            cout << _arr[i] << " ";
        cout << endl;
    }

    virtual ~basic_array_base() = default;

protected:

    T _arr[size];
};

template<typename T, int size>
class basic_array : public basic_array_base<T, size>  // acum clasa noastra
                                                      // template basic_array
                                                      // este doar o clasa
                                                      // template care
                                                      // mosteneste clasa
                                                      // template de baza/
                                                      // Nu vom scrie nimic
```

```
                                                    // in corpul acestei
                                                    // clase, deoarece
                                                    // vrem sa se comporte
                                                    // fix ca cea de baza
{
};

template<int size>                                  // aici avem o
                                                    // specializare partiala
                                                    // a clasei basic_array.
                                                    // Se observa ca doar
                                                    // functia print trebuie
                                                    // suprascrisa deoarece
                                                    // restul atributelor
                                                    // sunt preluate prin
                                                    // mostenire.
class basic_array<char, size> : public basic_array_base<char, size>
{
public:

    void print()
    {
        for (int i = 0; i < size; i++)
            cout << this->_arr[i];                  // din niste motive
                                                    // dubioase, aici daca
                                                    // nu puneam this->_arr
                                                    // nu mergea. Trebuie
                                                    // neaparat sa fie cu
                                                    // this->, nu _arr
                                                    // direct. Mai multe
                                                    // detalii aici:
        // https://stackoverflow.com/a/6592617 sau aici:
        // https://isocpp.org/wiki/faq/templates#nondependent-name-lookup-members
        cout << endl;
    }
};

int main()
{
    basic_array<int, 10> arr;

    for (int i = 0; i < 10; i++)
        arr[i] = i;

    arr.print();

    basic_array<char, 14> hlw;
    strcpy(hlw.get_array(), "Hello, World!");

    hlw.print();

    return 0;
}
```

In fiecare an pica cel
putin 50% la POO

prostii -0.1 pe prostie

fara prostii
0.1p

variabile publice = restanta

nu compileaza = restanta

Safe Exam Browser

1.2 Specializare parțială pentru pointeri

Fie o clasă template care are rolul de a păstra o copie a unui obiect de tipul dat ca argument template-ului. O astfel de clasă ar arăta astfel:

```
template<typename T>
class storage
{
public:

    storage(T obj) :
        _obj(obj)
    {
    }
```

```
    ~storage()
    {
    }

    void print()
    {
        cout << _obj;
    }

private:

    T _obj;
};
```

Clasa aceasta funcționează pentru orice obiect (presupunem că are constructorul de copiere scris corect și că are definit operator de afișare). Totuși, în cazul pointerilor, această clasă face o copie a pointerului, nu o copie a ce se află la adresa către care pointează. Noi probabil am vrea ca obiectul nostru să aibă alt pointer care pointează către un nou obiect, care este o copie a obiectului către care pointează argumentul dat în constructor. Deci, putem face o specializare parțială a clasei care să aibă acest comportament în cazul pointerilor. O astfel de specializare ar arăta astfel:

```
template<typename T>
class storage
{
public:

    storage(T obj) :
        _obj(obj)
    {
    }

    ~storage()
    {
    }

    void print()
    {
        cout << _obj;
    }

private:

    T _obj;
};

template<typename T>
class storage<T*>           // specializare pentru pointeri
{
public:

    storage(T* obj)
    {
        _obj = new T(*obj); // pastram o copie a obiectului catre care pointeaza
                            // pointerul obj.
```

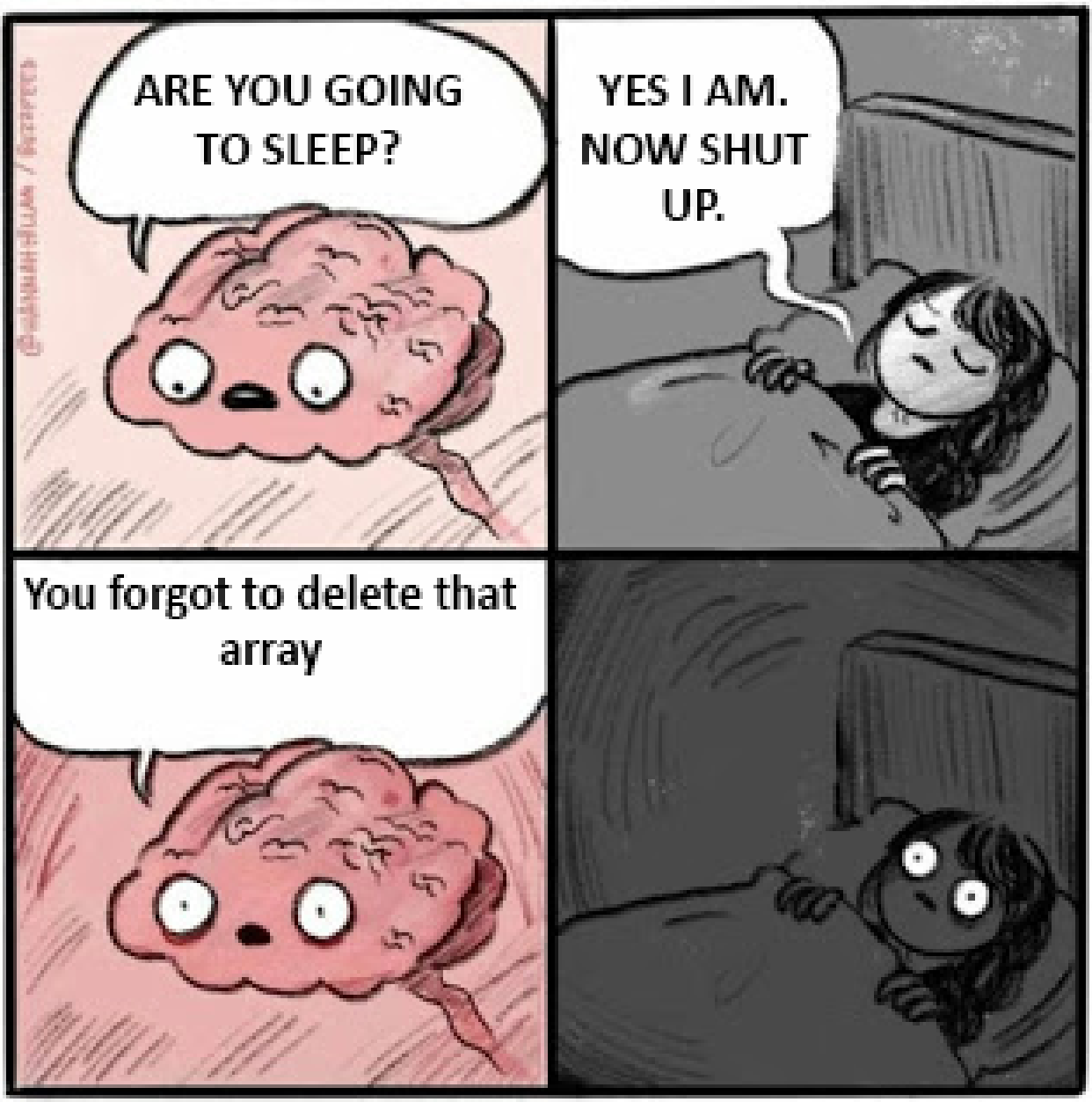
```
    }

    ~storage()
    {
        delete _obj;
    }

    void print()
    {
        cout << *_obj;
    }

private:

    T* _obj;
};
```


ARE YOU GOING TO SLEEP?
YES I AM. NOW SHUT UP.
You forgot to delete that array

Observi ca ai o variabila publica in codul de la colocviu dupa ce ai trimis codul.	
Vezi ca ai depasit termenul pentru trimis codul de la colocviu, deci oricum nu o sa se uite.	
Realizezi ca o sa ai restanta la POO	


Silviu
The chad C++
GOOD LORD!
WHAT IS HAPPENING IN THERE?
LEARNING C++

2 Keyword-ul auto

In C++ putem sa il lasam pe compilator sa deduca tipul de date al unei variabile. Acest lucru se numeste type inference.

```
#include <iostream>

using namespace std;

int add(int x, int y)
{
    return x + y;
}

auto add2(int x, int y)          // de evitat returnarea prin auto
{
    return x + y;
}

auto add3(int x, int y) -> int   // se poate si asta
{
    return x + y;
}

void add4(auto x, auto y)         // valid incepand cu C++20
{
    cout<<x+y;
}

int main()
{
    auto d = 7.0;   // va avea tip double
    auto i = 1+2;   // va avea tip int
    auto sum = add(5, 6);   // va avea tip int va asta returneaza add

    auto x;         // error: cannot deduce auto type

    auto sum2 = add2(1,2);

    add4(1, 2);

    add4(1, 6.0);
}
```

3 STL : standard template library

3.1 Containere (link)

3.2 Iteratori (link)

3.3 Algoritmi (link)

4 Intrebari

1. De ce vrem sa transmitem catre copy constructor prin referinta?

Pentru ca altfel se face infinite loop. Pentru detalii vezi Tutoriat 1.

2. De ce vrem sa transmitem catre copy constructor prin parametru const?

   Pentru ca vrem sa facem copii si din obiecte constante.

   Pentru ca vrem sa facem copii si din obiecte temporare.

   Pentru ca are sens sa nu modificam obiectul transmis pentru a fi copiat.

3. Care sunt cateva din diferentele dintre copy constructpr si operatorul=?

   La operatorul= trebuie sa ne amintim ca obiectul exista dinainte si poate avea memorie alocata ce trebuie dezalocata.

4. De ce am vrea sa avem un destructor virtual? Dati un exemplu.

   Cand dezalocam un obiect de tip derivata dintr-un pointer de tip baza. Pentru mai multe detalii vezi Tutoriat 4 : Destructor virtual.

5. De ce am vrea sa avem un destructor pur virtual?

   Daca vrem sa facem clasa abstracta dar nu avem nicio alta functie pe care o vrem pur virtuala.

6. Care sunt cateva particularitati ale destructorului pur virtual?

   Trebuie neaparat implementat in afara functiei daca vrem sa facem mostenire din acea clasa abstracta.

   Chiar daca in clasele derivate nu implementam destructorul, clasele derivate nu vor fi abstracte, pentru ca se foloseste cel dat de compilator.

7. Cum putem crea un copy constructor cu mai multi parametri?

   Nu se poate.

8. Ce e aia o functie inline?

   O functie implementata in clasa sau o functie care are keyword-ul inline.

9. Descrieti keywork-ul this.

   Stocheaza adresa obiectului implicit.

   Tipul sau de date este clasa * const, adica pointer de tip constant la un obiect.

   Daca obiectul este constant, atunci tipul sau de date este const clasa* const, adica pointer constant la obiect constant.

   Functiile prieten nu au pointerul this.

   Functiile statice nu au pointerul this.

10. Descrieti variabilele statice.

    Exista o singura copie a acelei date pentru toata clasa, adica pentru toate obiectele de tipul clasei.

    Trebuie initializate in afara clasei pentru a se aloca memorie pentru ele.

    Nu pot aparea in liste de initializare.

    Pot fi accesate atat de obiecte cat si de functii statice.

11. Descrieti metodele statice.

    Nu au pointerul this.

    Pot fi apelate si de obiect, si prin intermediul clasei.

    Au acces doar la datele si metodele statice din clasa.

    O functie nu poate avea o varianta statica si una nestatica.

    Nu pot fi virtuale pentru ca nu sunt legate de un obiect.

    Nu pot fi functii const pentru ca nu sunt legate de un obiect.

12. Care sunt operatorii care nu se pot supraincarca?

    ::, .*, . , ?:, sizeof, typeid

13. Cum putem pune operatori cu valori implicite?

    Nu putem. exceptie operatorul ().

14. Cum putem redefini numarul de operanzi ai unui operator?

    Nu putem.

15. Cum putem redefini precedenta si asociativitatea operatorilor?

    Nu putem.

16. Ce operatori se mostenesc si ce operatori nu se mostenesc in clasele derivate?

    Se mostenesc toti operatorii, in afara de operator=.

17. Care sunt operatorii care nu se pot supraincarca ca functii friend?

    Operatorii =, (), [] si -¿.

18. Descrieti operatorul de cast.

    Nu are tip returnat pentru ca tipul returnat e mereu acleasi cu numele operatorului.

    Nu poate avea parametri.

19. Descrieti tipurile de polimorfism.

    Polimorfism la compilare (supraincarcare de operatori si functii) si polimorfism la executie (virtual, RTTI, VTABLE).

20. Descrieti object slicing-ul.

21. Descrieti upcasting-ul.

22. Descrieti downcasting-ul.

5 Exercitii

Exercitiul 1

```
#include <iostream>
#include <vector>
using namespace std;
class Test {int i;
public:
    Test(int x = 0):i(x)
    {
        cout<<"C ";
    };
    Test(const Test& x)
    {
        i = x.i;
        cout<<"CC ";
    } ~Test(){cout<<"D ";}
};
int main()
{
    vector<Test> v;
    cout<<"\n";
```

```
    v.push_back(1);
    cout<<"\n";
    Test ob(3);
    cout<<"\n";
    v.push_back(ob);
    cout<<"\n";
    Test& ob2 = ob;
    cout<<"\n";
    v.push_back(ob2);
                cout<<"\n";
    return 0;
}
```

Exercitiul 2

```
#include <iostream>
using namespace std;
class A
{
    int valoare;
    public: A(int param1=3):valoare(param1){}
    int getValoare(){return this->valoare;}
};
int main()
{
    A vector[]={*(new A(3)),*(new A(4)),*(new A(5)),*(new A(6)) };
    cout<<vector[2].getValoare();
    return 0;
}
```

Exercitiul 3

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

class vector {
    int *p, nr;

public:
    operator int() { return nr; }
    vector(int);
};

vector::vector(int n)
{
    p = new int[n];
    nr = n;
    while (n--)
        p[n] = n;
}

void f(int i)
{
    while (i--)
        cout << i << endl;
```

```
}

void main()
{
    vector x(10);
    f(x);
}
```

Exercitiul 4

```
#include <iostream>

using namespace std;

class A
{
    int _x;
public:
    A(int x=9) : _x(x) {};
    virtual ~A() =0;
};
A::~A()
{

}

class B : public A
{
    int _y;
public:
    B(int y=5) : _y(y) {};
};

A f()
{
    B b;
    return b;
}

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