Tutoriat 6 POO

Bianca-Mihaela Stan, Silviu Stăncioiu

April 2021

1 Alti smart pointers

1.1 nft_ptr

Toată lumea știe că dacă adaugi blockchain pentru a rezolva o problemă, problema devine automat simplă, transparentă și sigură din punct de vedere criptografic. De aceea, niște oameni au inventat nft_ptr. Se comportă fix ca *unique_ptr*, doar că implementarea din spate folosește blockchain. De ce ai folosi *unique_ptr* care are o performanță de 0.005 secunde la executare, când poți folosi nft_ptr care are o performanță de 3

secunde? Mai multe despre acest tip de pointer aici:

https://github.com/zhuowei/nft_ptr

2 Alte cast-uri

2.1 const_cast

const_cast este un tip de cast care ne permite sa transform un pointer constant (adică pointer care nu are voie să modifice datele către care pointează) în pointer care nu este constant. Fie următorul exemplu:

```
#include <iostream>

using namespace std;

class test
{
public:
```

```
    test() :
        a(5)
    {
    }

    void f() const                          // functia este constanta, deci
                                            // in interiorul functiei, pointerul
                                            // this este constant. Adica are
                                            // tipul de date: const test*.
    {
        const_cast<test*>(this)->a = 10; // transformam pointerul this in
                                            // pointer care nu este constant,
                                            // iar acum avem acces sa modificam
                                            // datele care se afla in zona
                                            // in care pointeaza this.
    }

    int a;
};

int main()
{
    test t;

    cout << t.a << endl;                    // se afiseaza 5

    t.f();                                  // desi functia este constanta,
                                            // ea tot o sa modifice valori din
                                            // t

    cout << t.a << endl;                    // se afiseaza 10

    return 0;
}
```

Prin *const_cast* putem și să transmitem un pointer constant ca argument al unei funcții care primește ca argument un pointer normal. Exemplu:

```
#include <iostream>

using namespace std;

void fun(int* a)
{
    cout << *a;
}

int main(void)
{
    const int val = 10;                         // avem un obiect constant
    const int* ptr = &val;                      // luam un pointer constant
                                                // catre obiect
    int* regular_ptr = const_cast<int*>(ptr);   // transformam pointerul in
                                                // pointer care nu este
                                                // constant
```

```
    fun(regular_ptr);                           // acum putem apela functia,
                                                // iar pe ecran se va afisa
                                                // 10.

    return 0;
}
```

Atunci când folosim *const_cast* putem modifica valorile către care se pointează, dar trebuie să ne asigurăm că nu modificăm obiecte care au fost declarate de la bun început cu *const* (cum este variabila *val* din programul de mai sus). Dacă încercăm să modificăm obiecte care de la bun început au fost declarate ca fiind *const*, atunci vom avea undefined behaviour, adică nu știm cum se va comporta programul. Exemplu:

```
#include <iostream>

using namespace std;

void fun(int* a)
{
    *a = 20;     // nu stim cum se va comporta aceasta linie
}

int main(void)
{
    const int val = 10;
    const int* ptr = &val;
    int* regular_ptr = const_cast<int*>(ptr);

    fun(regular_ptr);
    cout << val; // nu stim ce se afiseaza pe ecran

    return 0;
}
```

În exemplul de mai sus avem undefined behaviour. Dacă val nu ar fi fost declarat cu *const*, atunci nu am fi avut undefined behaviour și s-ar fi afișat 20.

Aceste lucruri sunt similare și folosind C-style cast. Diferența este că în cazul lui *const_cast* se face și type checking (adică nu ne lasă să transformăm din pointer de *int* in pointer de *char* de exemplu).

2.2 reinterpret_cast

reinterpret_cast este folosit pentru a transforma dintr-un tip de pointer in altul fara a face nicio verificare, pur și simplu transformă direct.

3 template-uri

3.1 Function templates

Un function template este un pattern pentru crearea de functii.

```
#include <iostream>

using namespace std;

template <typename T>                   // asta ii spune compilatorului ca
```

```
                                        // T este un template parameter
const T& max(const T& x, const T& y)
{
    return (x > y) ? x : y;
}

int main()
{
    int x = max(3,7);                   // Aici compilatorul vede ca ii trebuie
                                        // max(int, int) si face o copie a functiei
                                        // template, cu int in loc de T.
    return 0;
}
```

Un alt exemplu de utilizare:

```
#include <iostream>

template <class T>
T average(T *array, int length)
{
    T sum(0);
    for (int count{ 0 }; count < length; ++count)
        sum += array[count];

    sum /= length;
    return sum;
}

int main()
{
    int array1[]={ 5, 3, 2, 1, 4 };
    std::cout << average(array1, 5) << '\n';        // Afiseaza 3

    double array2[]={ 3.12, 3.45, 9.23, 6.34 };
    std::cout << average(array2, 4) << '\n';        // Afiseaza 5.535

    return 0;
}
```

3.2 Template classes

Clasele template ne ajuta sa facem containere. Spre exemplu, vrem sa avem vectori de int-uri, vectori de un tip A creat de noi, etc. Putem sa facem o singura clasa template care sa ne permita sa cream vectori pentru fiecare dintre aceste tipuri.

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

template <class T>                          // Asa ii spunem compilatorului
                                            // ca asta e o clasa template.
class Array
{
private:
```

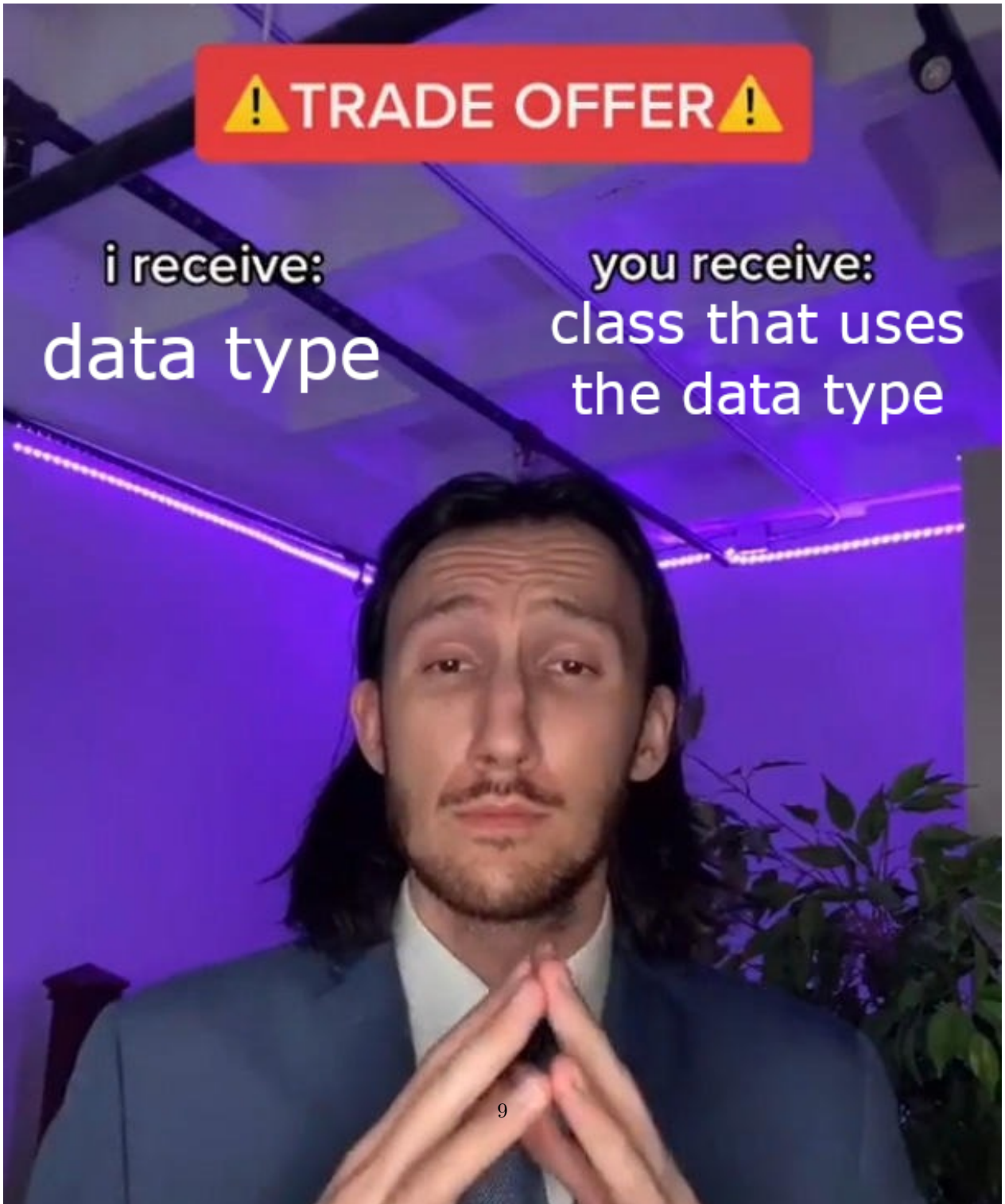
```
    int m_length{};
    T *m_data{};

public:

    Array(int length)
    {
        assert(length > 0);                     // assert da eroare daca nu se indeplineste conditia as
        m_data = new T[length]{};
        m_length = length;
    }

    Array(const Array&) = delete;               // Nu avem copy constructor.
    Array& operator=(const Array&) = delete;    // Nu avem operatorul =.

    ~Array()
    {
        delete[] m_data;
    }

    void erase()
    {
        delete[] m_data;                        // Dezalocam memoria.

        m_data = nullptr;
        m_length = 0;
    }

    T& operator[](int index)                    // Operator de indexare.
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() const;                      // Pe asta o definim mai jos.
};

// Functiile membre definite in afara clasei au nevoie de declaratia cu template.
template <class T>
int Array<T>::getLength() const                 // Observatie!!!
                                                // Numele clasei este Array<T> nu Array.
{
  return m_length;
}

class B
{
public:
    int _x;
    B(int x = 0) : _x(x) {}
};

int main()
{
```

```
    Array<int>  a(10);              // Aici se defineste (efectiv se scrie codul)
                                    // pentru o clasa Array in care inlocuim T cu int.
    for(int i=0; i<10; i++)
    {
        a[i]=i;
        cout<<a[i]<<" ";
    }
    cout<<"\n";
    Array<B> b(20);                 // La fel aici cu B.
    for(int i=0; i<20; i++)
    {
        b[i]=B(i);
        cout<<b[i]._x<<" ";
    }
    return 0;
}
```

Again, o clasa template nu e decat un pattern, nu e o clasa in sine.

Atentie! Pentru template classes trebuie sa puneti definitiile tuturor metoldelor in .h, nu o sa compileze daca le puneti in .cpp. De ce? Pentru ca prima data se ruleaza .h-urile, vede ca are nevoie de `Array<int>` si `Array<B>` dar pana ajunge sa compileze .cpp-ul, a uitat ca are de creat `Array<int>` si `Array<B>`.

templates be like:



3.3 Template non-type parameters

Pana acum am vazut cum sa definim o functie sau o clasa care primeste paramtri de un tip T, template. Insa avem situatii in care ne trebuie si niste parametri ficsi, nu de tip template.

```
#include <iostream>

template <class T, int size>                // size e non-type parameter
class StaticArray
{
private:
    T m_array[size];                        // Acel size controleaza dimensiunea
                                            // array-ului.
public:
    T* getArray();

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

template <class T, int size>                // Cum se defineste o functie pentru
                                            // o clasa cu non-type parameter.
T* StaticArray<T, size>::getArray()
{
    return m_array;
}

int main()
{
    StaticArray<int, 12> intArray;

    for (int count=0; count < 12; ++count)
        intArray[count] = count;

    for (int count=11; count >= 0; --count)
        std::cout << intArray[count] << " ";
    std::cout << '\n';

    StaticArray<double, 4> doubleArray;

    for (int count=0; count < 4; ++count)
        doubleArray[count] = 4.4 + 0.1*count;

    for (int count=0; count < 4; ++count)
        std::cout << doubleArray[count] << ' ';

    return 0;
}
```

3.4 Specializarea template-urilor

Atunci când lucrăm cu template-uri, de regulă scriem același comportament pentru fiecare tip de date cu care vom lucra. Totuși, pentru unele tipuri de date este posibil să fie nevoie să avem un comportament diferit.

De aceea, putem crea specializări ale clasei/ funcției template care să implementeze funcționalități diferite pentru anumite tipuri de date. Exemplu: presupunem că avem o funcție template de sortare care folosește *QuickSort* pentru orice tip de date pe care îl primește. Totuși, din motive de eficiență, am vrea ca dacă primește un array de caractere să folosească *Count Sort*. Putem face acest lucru cu o specializare a funcției pe tipul char. Exemplu:

```
#include <iostream>

using namespace std;

template<typename T>
void sort(T array[], int n)
{
    cout << "Folosim QuickSort" << endl;
}

template<>
void sort<char>(char array[], int n) // avem o specializare a functiei
                                     // pentru tipul char. Deci daca
                                     // functia se va apela cu un array
                                     // de char-uri, se va intra pe
                                     // aceasta specializare
{
    cout << "Folosim Count Sort" << endl;
}

int main()
{
    int a[100];
    char b[100];

    sort(a, 100);                    // va intra pe functia template de
                                     // baza
    sort(b, 100);                    // va intra pe specializarea cu char.

    return 0;
}
```

Observație: În programul de mai sus am apelat funcțiile direct, iar compilerul și-a dat seama pe ce specializări trebuie să intre. Am fi putut și să îi spunem explicit pe ce specializări vrem să intre, iar rezultatul ar fi fost același.

Am văzut că așa se pot specializa funcțiile template. Putem avea specializări și pentru clasele template. Exemplu:

```
#include <iostream>

using namespace std;

template<typename T>
class test
{
public:

    test()
    {
```

```
        cout << "Constructor general" << endl;
    }
};

template<>
class test<int>        // specializam clasa test pentru tipul de date int.
{
public:

    test()             // acum, cand cream un obiect de tipul test<int>
                       // se va apela acest constructor.
    {
        cout << "Constructor specializat pentru int" << endl;
    }
};

int main()
{
    test<char>      a; // se apeleaza constructorul pentru test<char>,
                       // adica cel de uz general. Se afiseaza mesajul
                       // din constructorul general.
    test<int>       b; // se apeleaza constructorul pentru specializarea
                       // int, deci se va afisa pe ecran ca este
                       // specializarea pentru int.
    test<test<int>> c; // tipul pe care il dam la template test
                       // test<int>, care este diferit de int, deci
                       // se va apela constructorul general.

    return 0;
}
```

Atunci când noi creăm o specializare pentru o funcție sau pentru o clasa, compilatorul ne crează o cu totul altă funcție/ clasă pentru tipul respectiv. Atunci când se decide pe care specializare să intre atunci când instanțiem un obiect sau apelăm o funcție, compilatorul va căuta prima oară o specializare cât mai apropiată de tipul de date pe care îl dăm, iar dacă nu găsește, va intra pe clasa/ funcția generală.

3.5 Specializări parțiale

Fie următoarea clasă template care primește un tip de date T și un număr întreg n și ține un array cu n obiecte de tipul T. Vom considera o funcție template *print* care afișează elementele unui astfel de array. Vom dori ca în cazul array-urile de caractere să afișeze fiecare caracter fără să pună spații între ele, iar în cazul altor tipuri de date, să la afișeze cu spații între ele. Putem avea o specializare atfel:

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

using namespace std;

template<typename T, int size>                  // avem un template pentru
                                                // un array de obiecte
                                                // de tipul T si lungime
                                                // 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>                  // avem o functie
                                                // template
                                                // care ne afiseaza un
                                                // astfel de array
                                                // pe caz general

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

template<>
void print(basic_array<char, 14>& arr)          // aici avem o
                                                // specializare a
                                                // functiei
                                                // pentru array-urile
                                                // de char cu 14 elemente
{
    for (int i = 0; i < 14; 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);                                // se vor afisa numerele
                                                // 0, 1, 2 separate prin
                                                // spatii

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

```
    print(str14);                                   // se va afisa sirul
                                                    // "Hello, world!" asa
                                                    // cum ne-am astepta sa
                                                    // se afiseze un sir,
                                                    // adica fara spatii
                                                    // intre caractere
                                                    // deoarece
                                                    // se intra pe
                                                    // sepcializarea
                                                    // basic_array<char, 14>

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

    print(str12);                                   // aici nu se mai intra
                                                    // pe specializare,
                                                    // deoarece lungimea
                                                    // este 12, nu 14. Deci
                                                    // se va afisa sirul
                                                    // cu spatii intre
                                                    // caractere.

    return 0;
}
```

După cum se observă, în cazul acesta nu are sens o astfel de specializare, pentru că funcționează doar pe array-uri de char de lungime 14. Noi am vrea să facem asta pe caz general, nu doar pe cazul în care avem array-uri de lungime 14. Pentru asta, vom folosi o specializare parțială, adică o specializare care este specializată doar pentru char, iar size-ul nu este specializat. Pentru asta, vom avea următorul program:

```
#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>                              // aici punem doar lucrurile
                                                // pe
                                                // care nu le specializam
void print(basic_array<char, size>& arr)        // se vede din argument ca
                                                // am specializat doar
                                                // primul
                                                // tip pe care il primeste
                                                // clasa, iar lungimea
                                                // ramane cea templetizata.
{
    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!"); // acum se afiseaza cum am
                                              // vrea si aici

    print(str12);

    return 0;
}
```

Deci, specializarea parțială înseamnă că noi specializăm doar o parte din template, nu în întregime.

4 Exercitii

Exercitiul 1

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

```
template<class X>void test(X &a, X &b)
{
    X temp;
    temp=a;
    a=b;
    b=temp;
    cout<<"ttest\n";
}
void test(int &c,int &d)
{
    int temp;
    temp=c;
    c=d;
    d=temp;
    cout<<"test 1\n";
}
void test(int e,int f)
{
    int temp;
    temp=e;
    e=f;
    f=temp;
    cout<<"test 2\n";
}
void test(int g,int *h)
{
    int temp;
    temp=g;
    g=*h;
    *h=temp;
    cout<<"test 3\n";
}
int main()
{
    int a=5,b=15,c=25,*d=&a;
    test(a,b);
    test(c,d);
    return 0;
}
```

Exercitiul 2

```
#include <iostream>
template <class T, class U>
T f(T x, U y)
{
    return y;
}
int f(int x, int y)
{
    return x - y;
}
int main()
{
    int *a = new int(3), b(23);
```

```
    std::cout << *f(a, b);
    return 0;
}
```

Exercitiul 3

```
#include <iostream>
template <class T, class U>
T f(T x, U y)
{
    return x+y;
}
int f(int x, int y)
{
    return x - y;
}
int main()
{
    int *a = new int(3), b(23);
    std::cout << *f(a, b);
    return 0;
}
```

Exercitiul 4

```
#include <iostream.h>
template <class T, class U>
T f(T x, U y)
{
    return x + y;
}
int f(int x, int y)
{
    return x - y;
}
int main()
{
    int *a = new int(3), b(23);
    cout << *f(a, b);
    return 0;
}
```

Exercitiul 5

```
#include <iostream>
using namespace std;
template <class T, class U>
T fun(T x, U y)
{
    return x + y;
}
int fun(int x, int y)
{
    return x - y;
}
```

```
int fun(int x)
{
    return x + 1;
}
int main()
{
    int *a = new int(10), b(5);
    cout << fun(a, b);
    return 0;
}
```

Exercitiul 6

```
#include <iostream>
using namespace std;
class A {
    int x;

public:
    A(int i = 0)
    {
        x = i;
        cout<<"A ";
    }
    ~A()
    {
        cout<<"~A ";
    }
    A operator+(const A& a) { return x + a.x; }
    template <class T>
    ostream& operator<<(ostream&);
};
template <class T>
ostream& A::operator<<(ostream& o)
{
    o << x;
    return o;
}
int main()
{
    A a1(33), a2(-21);
    cout << a1 + a2;
    return 0;
}
```