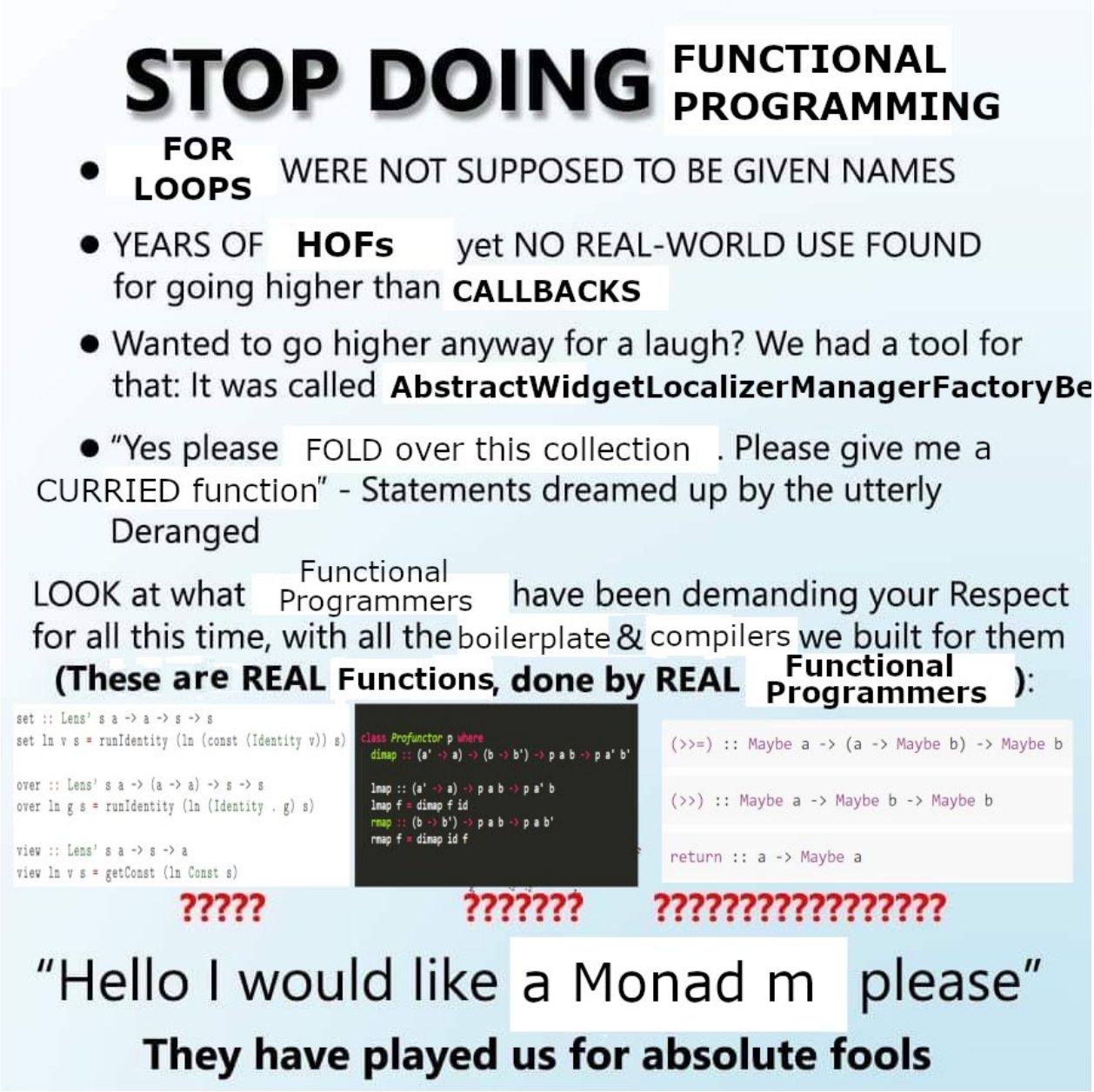
Tutoriat 8 POO

Bianca-Mihaela Stan, Silviu Stăncioiu

April 2021

Profesorii din FMI când introduc meme-uri în cursuri

1 Lambda functions



Putem face si in C++ functii anonime, asa cum faceati in Python. In C++ sintaxa e putin mai ciudata:

```
[ captureClause ] ( [ parameters ] ) [ ->  returnType ]
{
    statements;
}
```

Ce e intre paranteze [] e optional, adica captureClause, parameters si returnType sunt optionale.

Spre exemplu, o lambda triviala arata asa:

```
#include <iostream>

int main()
{
```

```
  []() {}; // defines a lambda with no captures, no parameters, and no return type

  return 0;
}
```

Sa luam un exemplu concret:

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <string>

using namespace std;

bool functie_normala (string str)
{
    // Metoda find este o metoda definita pe tipul string care cauta un sequence
    // intr-un string. Daca:
    // - sequence-ul este gasit, se returneaza un iterator la inceputul sau
    // - sequence-ul nu este gasit, se retuneaza string::npos
    return (str.find("nut") != string::npos);
}

int main()
{
    vector<string> arr{ "apple", "banana", "walnut", "lemon" };

    // Functia find_if este un function template definit in libraria algorithm care cauta primul elemen
    // care indeplineste o anumita conditie. Primeste ca parametri:
    // - un iterator la la punctul din care vrem sa incepem cautarea
    // - un iterator la unde sa se termine cautarea
    // - un predicat unar (care primeste un parametru), adica conditia pe care trebuie sa o
    // indeplineasca elementul pe care il cautam

    // Acel predicat poate sa fie o functie normala sau o functie anonima.

    // Cu functie normala.
    const auto found1 = find_if(arr.begin(), arr.end(), functie_normala);

    // Cu functie anonima.
    const auto found2 = find_if(arr.begin(), arr.end(),
                            [](string str) -> bool
                            {
                                return (str.find("nut") != string::npos);
                            });

    // Putem de asemenea sa instantiem o lambda pe care sa o folosim mai tarziu.
    auto finding_nut{
        [](string str) -> bool
                            {
                                return (str.find("nut") != string::npos);
                            }
    };

    const auto found3 = find_if(arr.begin(), arr.end(),
```

```
                            finding_nut);

    if (found3 == arr.end())
    {
        cout << "No nuts\n";
    }
    else
    {
        cout << "Found ";
        cout << (*found3) << '\n';
    }

    return 0;
}
```

1.1 Lambda functions si variabilele statice

Fie exemplul:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string>

using namespace std;

int main()
{
  // Afiseaza o valoare si de cate ori a fost apelata functia respectiva.
  auto print{
    [](auto value) {
        // Aceasta este o variabila statica dintr-o functie. La fel ca la
        // clase, ea se initializeaza o data si e incrementata cand se apeleaza
        // functia.
        static int callCount = 0;
        cout << callCount++ << ": " << value << '\n';
    }
  };

  // Observam ca deoarece parametrul functiei anonime print este auto, de
  // fiecare data cand se apeleaza functia cu alt tip de date se creeaza o noua "functie"
  // si deci se initializeaza o alta vriabila callCount.

  print("hello"); // Se afiseaza 0: hello
  print("world"); // Se afiseaza 1: world

  print(1); // Se afiseaza 0: 1
  print(2); // Se afiseaza 1: 2

  print("ding dong"); // Se afiseaza 2: ding dong

  return 0;
}
```

1.2 Capture clauses

Reluam exemplul de mai sus:

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <string_view>
#include <string>

using namespace std;

int main()
{
    vector<string> arr{ "apple", "banana", "walnut", "lemon" };

    // Vrem acum ca utilizatorul sa defineaza ce cautam in vector.
    cout << "search for: ";

    string search{};
    cin >> search;

    auto found1 = find_if(arr.begin(), arr.end(),
        [](string str) {
        // eroare: an enclosing-function local variable cannot be referenced in a lambda body unless it

        // Nu putem pur si simplu sa adaugam search la lista de parametri, pentru ca predicatul nostru
        return (str.find(search) != string_view::npos); // Error: search not accessible in this scope
    });

    auto found2 = find_if(arr.begin(), arr.end(),
        [search](string str) {
        // REZOLVARE
        return (str.find(search) != string_view::npos); // Error: search not accessible in this scope
    });


    if (found1 == arr.end())
    {
        cout << "Not found\n";
    }
    else
    {
        cout << "Found " << *found1 << '\n';
    }

    return 0;
}
```

1.3 Modificarea elementelor din capture clause

1.3.1 Modificarea copiilor din functie

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

```cpp
int main()
{
  int ammo = 10;

  auto shoot{
    // Se adauga mutable dupa lista de parametri ca sa putem sa si modificam
    // chestiile din clauza.
    // Modificarea nu afecteaza si variabila din main, ci doar copia care se
    // creeaza cand se apeleaza functia.

    // Se afiseaza:
    // Pew! 9 shot(s) left.
    // Pew! 8 shot(s) left.
    // 10 shot(s) left
    [ammo]() mutable {

      --ammo;

      cout << "Pew! " << ammo << " shot(s) left.\n";
    }
  };

  shoot();
  shoot();

  cout << ammo << " shot(s) left\n";

  return 0;
}
```

1.3.2 Modificarea prin referinta

```cpp
#include <iostream>

using namespace std;

int main()
{
    int ammo = 10;
    auto shoot{
            //Acum e transmis prin referinta, nu mai avem nevoie de mutable.
            [&ammo]() {
            // Schimbarile vor fi vizibile si in afara functiei.

            // Se afiseaza:
            // Pew! 9 shot(s) left.
            // 9 shot(s) left
            --ammo;

            cout << "Pew! " << ammo << " shot(s) left.\n";
            }
    };

    shoot();
```

```
    cout << ammo << " shot(s) left\n";

    return 0;
}
```

1.4 Default captures

```
#include <vector>

using namespace std;

int main()
{
    int health{ 33 };
    int armor{ 100 };
    vector<int> enemies{};

    // Capture health and armor by value, and enemies by reference.
    [health, armor, &enemies](){};

    // Capture enemies by reference and everything else by value.
    [=, &enemies](){};

    // Capture armor by value and everything else by reference.
    [&, armor](){};

    // Illegal, we already said we want to capture everything by reference.
    [&, &armor](){};

    // Illegal, we already said we want to capture everything by value.
    [=, armor](){};

    // Illegal, armor appears twice.
    [armor, &health, &armor](){};

    // Illegal, the default capture has to be the first element in the capture group.
    [armor, &](){};
}
```

2 Design patterns

Design pattern-urile sunt convenții de implementare a anumitor lucruri des întâlnite în programare. La teme și la colocviu e bine să folosiți design patterns.


Programmer
Multiple instances of a class
Singleton with private constructor and no copy constructor/ assignment operator

2.1 Singleton

Să presupunem că avem un joc, iar acel joc are o clasă *timer* care are scopul de a ne spune cât timp a trecut de la ultimul frame. Noi am avea nevoie de un singur astfel de obiect, deoarece informația pe care o poate da acest obiect este aceeași oriunde în program. Sigur pentru asta, am putea face o clasă statică cu acest rol, deși nu este plăcut să lucrăm cu clase statice, și nici nu este recomandat.

Singletone-ul este un design pattern care se folosește atunci când în întregul program ne dorim să avem o singură instanță a unei clase, o instanță globală. Pentru asta, de regulă se face o clasă cu un constructor privat și se păstrează un pointer static către un obiect de tipul acestei clase. Acest obiect poate fi accesat doar printr-o metodă statică *get_instance* din interiorul clasei. Pentru ca obiectul să nu poată fi copiat, vom șterge operatorul de atribuire și constructorul de copiere din clasă. Exemplu de singletone pentru un timer:

```
#include <iostream>
#include <ctime>

using namespace std;

class timer
{
public:

    timer (const timer&)                  = delete; // stergem
                                                    // constructorul de
                                                    // copiere pentru a
                                                    // ne asigura ca
                                                    // exista o singura
                                                    // instanta a
                                                    // acestui obiect

          void   operator=(const timer&) = delete; // stergem si
                                                    // operatorul de
                                                    // assignment
                                                    // din acelasi
                                                    // motiv pentru care
                                                    // am sters
                                                    // constructorul de
                                                    // copiere

    static timer* get_instance()                    // functia noastra
                                                    // statica care ne
                                                    // returneaza
                                                    // singura instanta
                                                    // a obiectului
    {
        if (!g_instance)                            // functia este lazy,
            g_instance = new timer();               // adica ea creeaza
                                                    // un obiect doar in
                                                    // momentul in care
                                                    // este nevoie.

        return g_instance;
    }
```

```
    static void   free_memory()                     // nu vrem sa avem
    {                                               // memory leak-uri,
        delete g_instance;                          // deci vom face
        g_instance = nullptr;                       // o functie care
    }                                               // da delete la
                                                    // instanta creata.

           void   update()                          // intr-un joc
    {                                               // s-ar apela aceasta
        _prev_time = _current_time;                 // functie in fiecare
        _current_time = clock();                    // frame pentru a
    }                                               // determina timpul
                                                    // scurs intre
                                                    // frame-uri

           float  get_delta_time()                  // functie care
    {                                               // ne returneaza
        float diff = _current_time - _prev_time;    // timpul scurs intre
        return diff / 1000.0f;                      // frame-uri in
                                                    // secunde
    }

private:

    timer()                                         // avem constructor
    {                                               // privat pentru
        _prev_time    = clock();                    // a putea sa facem o
        _current_time = clock();                    // instanta a clasei
    }                                               // doar in interiorul
                                                    // unei metode statice
                                                    // din clasa
private:

           clock_t _prev_time;
           clock_t _current_time;

    static timer*  g_instance;                      // pointerul catre
                                                    // obiectul global
};

timer* timer::g_instance = nullptr;                 // initializam
                                                    // pointerul catre
                                                    // instanta cu
                                                    // nullptr. Daca
                                                    // scrieti cod
                                                    // si in .h si .cpp
                                                    // mutati linia asta
                                                    // la inceputul
                                                    // cpp-ului pentru
                                                    // clasa (dupa
                                                    // include-uri si
                                                    // using namespace-
                                                    // uri)
```

```
void game_logic()                                       // are acces la
                                                        // la obiectul
                                                        // de tip timer
{
    cout << "Delta time: " << timer::get_instance()->get_delta_time() << endl;
}

int main()
{
    bool game_running = true;
    int iter = 0;

    while (game_running)
    {
        timer::get_instance()->update();

        game_logic();

        for (int i = 0; i < 10000; i++)                 // putin busy waiting
                                                        // pentru a nu ne
                                                        // afisa mereu
                                                        // 0 pe ecran.
            if (i % 10000 == 9999)
                cout << "";

        iter++;
        if (iter >= 10)
            game_running = false;
    }

    timer::free_memory();                               // nu uitam sa
                                                        // eliberam memoria
                                                        // alocata

    return 0;
}
```

Așa se implementează de regulă o clasă Singleton în C++. Se poate defini o clasă template $Singletone < T >$ care să facă un Singletone pentru tipul T (The exercise is left for the reader). În mod alternativ, dacă nu vrem să ne complicăm să avem un pointer pentru instanța noastră putem declara o variabilă statică de tipul clasei în interiorul funcției *get_instance* și să returnăm direct un pointer către ea. Exemplu:

```
static singleton* get_instance()
{
    static singleton instance;

    return &instance;
}
```

Se observă că în acest caz nu mai avem nevoie de o metodă $free\_memory$. Sincer, metoda asta de a face un singletone este mai ușoară, dar la noi la facultate se predă metoda cu pointer și probabil pe aceea vor să o vadă laboranții la colocviu/ teme... so... o folosiți pe aia. Asta o folosiți dacă o să vă trebuiască sa faceți singletone-uri în afara cursului de POO.

2.2 Factory

Fie un program în care avem o clasă de bază și câteva clase derivate (polimorfism). Noi ne dorim ca în funcție de o variabilă să creăm un obiect pe care să îl ținem într-un pointer de tipul clasei de bază. Putem face ceva ceva de genul:

```
#include <iostream>

using namespace std;

class regular_dog
{
public:

    virtual ~regular_dog()
    {
    }

    virtual void bark()
    {
        cout << "Woof!" << endl;
    }
```

```
};

class fancy_dog : public regular_dog
{
public:
    void bark() override
    {
        cout << "May I woof, sir?" << endl;
    }
};

class random_dog : public regular_dog
{
public:
    void bark() override
    {
        cout << "random woof" << endl;
    }
};

int main()
{
    int type = 0;

    cin >> type;

    if (type > 2 || type < 0)
        return -1;

    regular_dog* dog = nullptr;

    if (type == 0)                  // avem if-uri pentru a determina
        dog = new regular_dog();    // tipul de obiect pe care il cream
    else if (type == 1)
        dog = new fancy_dog();
    else
        dog = new random_dog();

    dog->bark();

    delete dog;
    dog = nullptr;

    return 0;
}
```

Acest approach merge, deși, este posibil ca noi în program să fim nevoiți să facem asta de mai multe ori. Dacă tot facem copy-paste la if-else-uri nu e bine, pentru că dacă adăugăm o clasă nouă trebuie să modificăm peste tot unde avem seria asta de if-else-uri. Așa că, vom crea o clasă auxiliară care să ne creeze un obiect nou de un tip specificat. Vom avea:

```
#include <iostream>

using namespace std;
```

```
class regular_dog
{
public:

    virtual ~regular_dog()
    {
    }

    virtual void bark()
    {
        cout << "Woof!" << endl;
    }
};

class fancy_dog : public regular_dog
{
public:
    void bark() override
    {
        cout << "May I woof, sir?" << endl;
    }
};

class random_dog : public regular_dog
{
public:
    void bark() override
    {
        cout << "random woof" << endl;
    }
};

class dog_factory                                          // clasa noastra
                                                           // care se ocupa
                                                           // de crearea
                                                           // obiectelor
                                                           // de tip caine
{
public:

    enum dog_types                                         // avem un enum
    {                                                      // pentru tipurile
        regular_dog_type,                                  // claselor
        fancy_dog_type,                                    // pentru a ne
        random_dog_type                                    // usura treaba
    };

    static regular_dog* create_dog(dog_types dog_type)     // functia care
                                                           // creaza un obiect
                                                           // nou dintr-o
                                                           // clasa data
    {
        if (dog_type == dog_types::regular_dog_type)
            return new regular_dog();
```

```
        else if (dog_type == dog_types::fancy_dog_type)
            return new fancy_dog();
        else if (dog_type == dog_types::random_dog_type)
            return new random_dog();

        return nullptr;
    }
};

int main()
{
    int type = 0;

    cin >> type;

    if (type > 2 || type < 0)
        return -1;

    regular_dog* dog = dog_factory::create_dog((dog_factory::dog_types)type);
                                                                    // acum nu mai
                                                                    // este nevoie
                                                                    // sa scriem
                                                                    // if-urile
                                                                    // mereu cand
                                                                    // avem nevoie
                                                                    // de a crea
                                                                    // un obiect
                                                                    // dandu-se
                                                                    // tipul
                                                                    // intr-o
                                                                    // variabila

    dog->bark();

    delete dog;
    dog = nullptr;

    return 0;
}
```

Se observă că acest approach este mult mai bun decât o înșiruire de if-uri oriunde avem nevoie de asta. Acum, dacă se introduce o clasă nouă, doar se adaugă un element nou in enum și se adaugă încă un *if* în funcția din factory.

Boboc: incep sa cred ca Silviu tine tutoriat doar ca sa vorbeasca despre crypto, GME si sa isi promoveze jocul.

Silviu: