Facultatea de Matematică și Informatică,
Universitatea din București

# Tutoriat 3

Programare Orientată pe Obiecte - Informatică, Anul I

Tudor-Gabriel Miu
Radu Tudor Vrînceanu
3-29-2024

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

# Cuprins

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

# Fundamentele Programării Orientate pe Obiecte în C++

## Lista de inițializare (continuare)

Data trecută, am discutat în preponderent despre constructori, destructori și alte concepte despre clase (i.e. locale, prietene, etc…). În această categorie vom descrie ce reprezintă lista de inițializare și de ce o folosim. Lista de inițializare reprezintă o parte esențială a constructorului întrucât prin aceasta putem **apela constructorii claselor tip părinte** (în cazul moștenirii, vom discuta despre acest concept în următoarele tutoriate) sau **putem inițializa valori ale atributelor** fără a ne folosi de corpul constructorului.

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

class Person {
private:
    char *name;
    int age;
public:
    Person(int age = 0, const char* name = NULL) : age(age), name(new
char[strlen(name) + 1]) {
        for (int i = 0, j = strlen(name); i < j; i++) {
            this->name[i] = name[i];
        }
    }

    ~Person() {
        if (name != NULL) {
            delete[] name;
        }
    }

    char* getName() const {
        return name;
    }
};

int main() {
    Person p(20, "John Doe");
    cout << p.getName() << endl;

    return 0;
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

În exemplul de mai sus în lista de inițializare a constructorului Person am setat atributul age (oferindu-i o valoare) și name (oferindu-i o adresă de început a array-ului). Acum vrem să vedem un alt caz esențial de utilizare a listei de inițializare în legătură cu variabilele constante dintr-o clasă. Vom lua următoarea situație:

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

class Person {
private:
    char *name;
    int age;
    const char* CNP;
    const int parents;
public:
    Person(int age = 0, const char* name = NULL, const char* CNP = NULL, int
parents = 1) : age(age), name(new char[strlen(name) + 1]) {
        for (int i = 0, j = strlen(name); i < j; i++) {
            this->name[i] = name[i];
        }
        this->CNP = new char[strlen(CNP) + 1];
        for (int i = 0, j = strlen(CNP); i < j; i++) {
            this->CNP[i] = CNP[i];
        }
        this->parents = parents;
    }

    ~Person() {
        if (name != NULL) {
            delete[] name;
        }
        if (CNP != NULL) {
            delete[] CNP;
        }
    }

    char* getName() const {
        return name;
    }

    const char* getCNP() const {
        return CNP;
    }
};

int main() {
    Person p(20, "John Doe", "2131415167002", 2);
    cout << p.getName() << ' ' << p.getCNP() << endl;

    return 0;
}
```

Ce problemă avem cu codul de mai sus? Observăm că deși constructorul este primul lucru care se apelează la instanțierea clasei, atributele de tip const nu pot să

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

ia valorile pe care noi le acordăm de ce? Întrucât lista de inițializare este defapt prima componentă a constructorului care se apelează, dacă un atribut este acolo atunci se atribuie valoarea dintre paranteze, iar dacă nu se ia o valoare default, în cazul pointerilor poate fi NULL, în cazul valorilor poate fi 0 sau altă valoare reziduală. Deci este important ca **atributele de tip const să primească valori în cadrul listei de inițializare**.

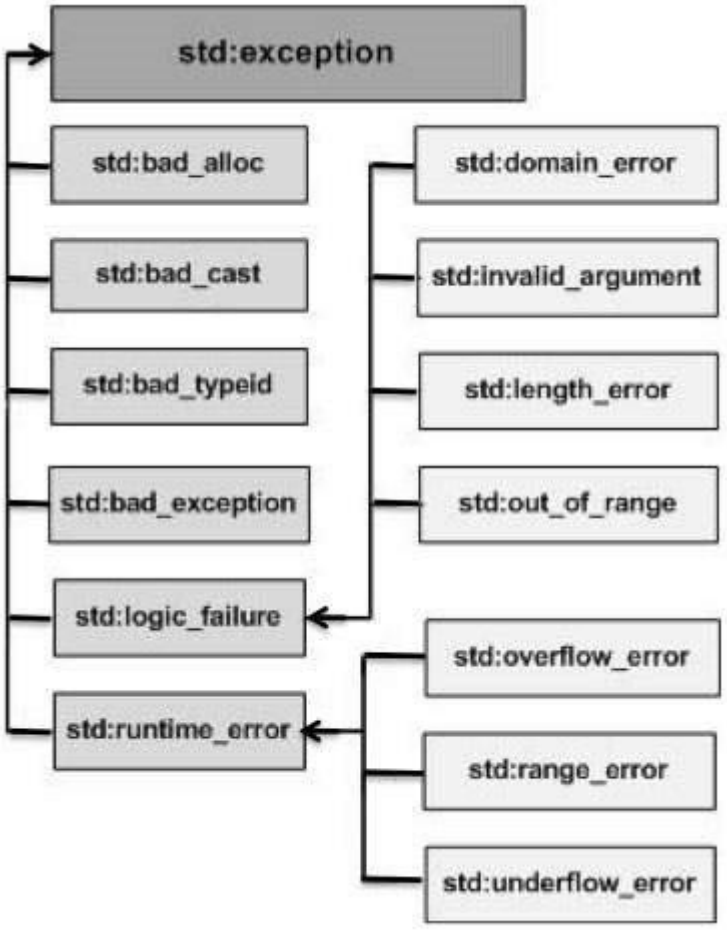
```
#include <iostream>
#include <cstring>
using namespace std;

class Person {
private:
    char *name;
    int age;
    const char* CNP;
    const int parents;
public:
    Person(int age = 0, const char* name = NULL, const char* CNP = NULL, int
parents = 1) : age(age), name(new char[strlen(name) + 1]), CNP(CNP),
parents(parents) {
        for (int i = 0, j = strlen(name); i < j; i++) {
            this->name[i] = name[i];
        }
    }

    ~Person() {
        if (name != NULL) {
            delete[] name;
        }
    }

    char* getName() const {
        return name;
    }

    const char* getCNP() const {
        return CNP;
    }
};

int main() {
    Person p(20, "John Doe", "2131415167002", 2);
    cout << p.getName() << ' ' << p.getCNP() << endl;

    return 0;
}
```

Aceasta este varianta corectă. Observăm totuși o diferență, de ce destructorul nu mai trebuie să apeleze operatorul delete[] pentru CNP? **R:** https://stackoverflow.com/a/59597680

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

# Excepții

Ce sunt excepțiile? Reprezintă un mecanism care se găsește în orice limbaj de programare high-level pentru tratarea cazurilor care sunt considerate anormale. În limbajul de programare C++, o excepție este o situație anormală care apare în timpul execuției programului și care întrerupe fluxul normal al acestuia. Excepțiile sunt folosite pentru a gestiona și trata erorile sau condițiile excepționale într-un mod controlat și elegant. Putem definii rpropriile noastre excepții, întrucât ele la bază sunt clase, iar standardul C++ oferă la dispoziție deja niște excepții de bază.




Pentru a putea trata o excepție ne folosim de blocuri de tip try & catch, în blocul try punem codul care ar putea arunca excepții, iar dacă este cazul în **blocul / blocurile** catch tratam individual, în funcție de excepție problemele.

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

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
void f(int x) throw (runtime_error, exception, int) {
    if (!(x % 15)) {
        throw runtime_error("FIZZBUZZ");
    } else if (!(x % 5)) {
        throw exception();
    } else if (!(x % 3)) {
        throw x;
    }
}

int main() {
    try {
        f(45);
    } catch (exception& e) {
        cout << "Am prins o exceptie din clasa exception " << e.what() <<
endl;
    } catch(...) {
        cout << "Prin ... prind toate tipurile de exceptii";
    }
    return 0;
}
```

Secvența de după antetul lui f și anume partea cu throw ne avertizează faptul că funcția poate arunca excepții, iar în paranteză ce excepții poate arunca respectiva funcție. **DEPRECATED DUPĂ C++11, DECI AVEȚI GRIJĂ LA ACEST LUCRU!**

```
main.cpp:5:15: warning: dynamic exception specifications are deprecated in C++11 [-Wdeprecated]
    5 | void f(int x) throw (runtime_error, exception, int) {
      |               ^~~~~
```

De ce runtime_error, intră pe catchul lui exception? Întrucât runtime_error, așa cum este arătat și în diagrama de mai sus este o clasă derivată (i.e. copilul) a clasei exception și putem apela la polimorfism pentru a putea face acest lucru (vom vedea mai târziu în seria de tutoriate mecanismul pe care se bazează polimorfismul). Putem să ne definim propriile noastre excepții moștenind clasa exception și implementând metoda what() din aceasta.

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

class SimpleException : public exception {
    const char* what() const noexcept {
        return "Simple exception";
    }
};

void f(int x) {
    if (!(x % 15)) {
        throw runtime_error("FIZZBUZZ");
    } else if (!(x % 5)) {
        throw exception();
    } else if (!(x % 3)) {
        throw SimpleException();
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
    }
}

int main() {
    try {
        f(21);
    } catch (exception& e) {
        cout << "Am prins o exceptie din clasa exception " << e.what() <<
endl;
    } catch(...) {
        cout << "Prin ... prind toate tipurile de exceptii";
    }
    return 0;
}
```

Putem bineînțeles să complicăm această excepție pentru a lua diverse informații unde s-a întâmplat excepția prin adăugarea unui constructor.

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

class MessageException : public exception {
private:
    string msg, code_scope;
public:
    MessageException(const string& code_scope, const string& message) :
msg(message), code_scope(code_scope) {}
    ~MessageException() noexcept {}
    const char* what() const noexcept {
        static string error_text = "[MessageException] Raised at: " +
code_scope + " with: " + msg;
        return error_text.c_str();
    }
};

void f(int x) {
    if (!(x % 15)) {
        throw runtime_error("FIZZBUZZ");
    } else if (!(x % 5)) {
        throw exception();
    } else if (!(x % 3)) {
        throw MessageException("f", "Received a FIZZ term.");
    }
}

int main() {
    try {
        f(21);
    } catch (exception& e) {
        cout << e.what() << endl;
    } catch(...) {
        cout << "Prin ... prind toate tipurile de exceptii";
    }
    return 0;
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

De asemenea ca să prindem doar excepția de tip MessageException putem să o punem într-un bloc de catch separat:

```
int main() {
   try {
       f(21);
   } catch (MessageException& e) {
       cout << e.what() << endl;
   } catch (exception& e) {
       cout << e.what() << endl;
   } catch(...) {
       cout << "Prin ... prind toate tipurile de exceptii";
   }
   return 0;
}
```

Există posibilitatea **de a da throw în catch și de a avea blocuri imbricate de try…catch**, ce este important de reținut este că **pentru try-ul respectiv se va lua catch-ul cu tipul respectiv sau cel moștenit în cazul claselor**. Există posibilitatea de a da throw fără nimic (int, double, clasă, etc…) acesta poate fi prins doar cu ajutorul catch-ului care prinde orice, anume catch(...).

# Funcții anonime (lambda expresii) și tipul auto

Funcțiile anonime (lambda expresiile) în C++ reprezintă funcții care pot fi declarate “in-place” și au un mod mai special de a fi scrise, ele ne pot ajuta pentru diverse chestii care pot fi folosite împreună cu biblioteca algorithm din STL (sortare, etc…).

```
/*
   [scop_extern](lista_parametrii) mutable noexcept/exception -> tip_returnat
{
       // corpul functiei anonime
       return data_tip_returnat;
   }

   scop_extern poate introduce in functia noastra anonima variabile declarate
anterior, dar care nu servesc ca parametrii, neaparat
*/
auto f = []() mutable noexcept -> const char* {
   return "Hello World!";
};
cout << f() << endl;
```

În C++, auto este un cuvânt cheie introdus în standardul C++11 care permite deducerea automată a tipului unei variabile în funcție de valoarea cu care este inițializată. Este folosit pentru a face codul mai concis și mai ușor de citit, evitând explicit specificarea tipului unei variabile. Atunci când folosești auto pentru declarația unei variabile, compilatorul deduce tipul acelei variabile pe baza tipului expresiei cu care este inițializată.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

# Exerciții

Cerința exercițiilor este de a determina dacă programele compilează sau nu, în cazul în care compilează ce trebuie să afișeze, iar în cazul în care nu compilează linia la care se produce eroarea și explicarea, respectiv rezolvarea ei modificând doar o linie în cod.

```cpp
// 1
#include <iostream>
using namespace std;

class A {
   int y;
public:
   A(int x = 2020) : y(x) { cout << "A"; }
   ~A() { cout << "~A"; }
   ostream& operator<<(ostream& out) const {
       out << this->y; return out;
   }
};

int main() {
   A a(2), b;
   a << (b << cout);
   return 0;
}
```

```cpp
// 2
#include <iostream>
using namespace std;

class C {
   int * const p;
public:
   C(int x) : p(&x) {(*p) += 3;}
   void set (int x) {p = &x;}
   friend ostream& operator<<(ostream& o, C x) {
       o << *x.p; return o;
   }
};

int main() {
   cout << C(3);
   return 0;
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```cpp
// 3
#include <iostream>
using namespace std;

class Person {
   static int generator;
   const int code;
   int age;
public:
   explicit Person(int cod = ++generator, int varsta = 18) : code(cod),
age(varsta) {
       cout << "Constructor Person ";
       cout << "cu codul " << code << " si varsta "<< age << '\n';
   }

};

int main() {
   Person person;

   return 0;
}
```

```cpp
// 4
#include <iostream>
struct Integer {
   int x;
   Integer(const int val = 0) : x(val) {}
   friend Integer operator+ (Integer& i, Integer& j) {
       return Integer(j.x + i.x);
   }
   friend std::ostream& operator<<(std::ostream& o, Integer i) {
       o << i.x; return o;
   }
};
int main() {
   Integer i(25), j(5), k(2020);
   std::cout << (i + j + k);
}
```

```cpp
// 5
#include <iostream>
int foo(int x, int y = 0) {
   return x + y - 2020;
}
int foo (int x) {
   return x + 2020;
}

int main() {
   std::cout << foo(5);
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```cpp
// 6
#include <iostream>

struct Integer {
   int x;
   Integer(const int val = 0) : x(val) {}
   Integer operator+(Integer& i) {
       return Integer(x + i.x);
   }
   friend std::ostream& operator<<(std::ostream& o, Integer& i) {
       o << i.x; return o;
   }
};

int main() {
   Integer i(25), j(5), k(2020);
   std::cout << (i + j + k);
}
```

```cpp
// 7
#include <iostream>
using namespace std;

class C {
   const int i;
public:
   C (int j = 2022) { this->i = j; }
   operator int () { return this->i; }
};

int main() {
   C c1(5), c2;
   cout << c1 << c2 << endl;
}
```

Soluții:

1. Programul compilează fără probleme, ilustrează forma non-”friend” a operatorului <<, imaginați-vă cum l-ați apela sub formă de metodă nu ca operator și vedeți ordinea parametrilor, ar fi ceva de genul b.operator<<(cout); Afișează AA20202~A~A.
2. Programul nu compilează avem un pointer constant și există o metodă de tip setter în clasa noastră care este **neconstantă** și impactează nu valoarea, ci adresa (i.e. dorește să modifice adresa către care pointează), cum pointerul este constant nu se poate acest lucru. Eroarea va fi la linia declarării setter-ului, iar pentru repararea ei eliminăm const-ul din declararea atributului p, îl vom lăsa doar int* p.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

3. Programul nu compilează deoarece variabila statică nu este inițializată. Pentru repararea ei va trebui să punem următoarea linie de cod int Person::generator = 0; în afara clasei.
4. Programul nu compilează deoarece return-ul by value a lui (i + j) îl va face ca și tip pentru referință const Integer, deci eroare se va genera la linia declarării operator+, iar rezolvarea ei constă în schimbarea tipului parametrilor Integer operator+(const Integer& i, Integer& j)
5. Programul nu compilează deoarece nu știe ce formă a funcției **OVERLOADED** foo să ia, rezolvarea ei constă în elimiarea valorii implicite a lui y din antetul funcției foo: int foo(int x, int y)
6. Aceeași problemă ca la 4., doar că de această dată pentru operatorul <<.
7. Programul nu compilează, trebuie ca toate atributele de tip const să fie plasate în lista de inițializare pentru inițializare, nu în corpul constructorului.