Seminarul II

1 Pointerul this

In fiecare dintre constructorii, destructorii si metodele clasei avem un pointer care indica către adresa obiectul curent, numit **this** . Scopul acestui pointer este de a putea menționa explicit ca ne referim la o metoda sau la o proprietate a obiectului curent.
In C++, folosirea pointerului **this** nu este necesara pentru a putea accesa o proprietate a obiectului curent, compilatorul va deduce din context daca ne referim sau nu la o proprietate a obiectului. In alte limbaje de programare (e.g. Java, C#) folosirea pointerului **this** este obligatorie.

```
class C {
    int x;
    public:
        C(int x) {
            this->x = x;
        }
};
```

2 Declarări inline

Atunci când declaram o clasa avem doua posibilități pentru a oferi implementarea constructorilor, destructorilor si a metodelor:

- in interiorul declarării clasei;
- in exteriorul declarării clasei;

Implementarea in interiorul declarării clasei se mai numește si implementare **inline**. Acest tip de implementare spune compilatorului ca acea zona de cod trebuie sa fie executata cat mai rapid posibil. Compilatorul va încerca sa încarce implementarea in memoria cache a procesorului.
Implementarea in exteriorul declarării clasei presupune ca fiecare constructor/destructor/metoda trebuie sa aibă o signatura declarata in interiorul clasei:

```
    [<tip_retur>] <nume_symbol>([<lista_parametrii>]);
```

Pentru a putea implementa in afara clasei trebuie sa ne referim la simbolul implementat folosind operatorul de rezoluție de scop **::** :

```
[<tip_retur>] <nume_clasa>::<nume_simbol>([<lista_parametrii>]) {
    /* implementarea propriu-zisa */
}
```

Implementarea in interiorul clasei nu este singura metoda de a implementa inline. Putem implementa metode/constructori/destructori inline si in afara clasei folosind cheie **inline** (inline explicit):

```
inline [<tip_retur>] <nume_clasa>::<nume_simbol>([<lista_parametrii>]) {
    /* implementarea propriu-zisa */
}
```

Exemplu:

```
class Stack {
    int *stack;
    unsigned size, tos;

    public:
        // constructorul fara parametrii implementat implicit inline
        Stack () {
            size = 10;
            stack = new int[size];
            tos = -1;
        }

        // Signatura constructorul cu parametrii,
        // signatura metodei de adaugare element in stiva si
        // signatura destructorul care vor implementata te in afara clasei
        Stack (int);
        void push(int);
        ~Stack();
};

inline Stack::Stack (int n) { // implementare inline explicita
    size = n;
    stack = new int[size];
    tos = -1;
}

void Stack::push (int x) {
    if (tos == size - 1) {
        return;
    }
    stack[++tos] = x;
}

inline Stack::~Stack () {  // implementare inline explicita
    delete[] stack;
    size = 0;
    tos = -1;
}
```

Deoarece declararea unei funcții inline este doar o sugestie către compilator, chiar daca noi nu marcăm o metodă ca inline, compilatorul va incerca să facă inline dacă poate. Pe scurt, compilatorul va încerca sa transforme inline cât mai mult din codul nostru, chiar dacă noi nu menționăm explicit asta.

3 Separarea declararii

În programe care au multe linii de cod si multe clase/functii, folosirea unui singur fișier pentru a salva codul duce la o gestionare greoaie a codului sursa. De aceea de fiecare dată cand declarăm o clasă, e bine sa o declarăm separat de restul codului.
Pentru a declara o clasa vom folosi:

- un fișier header (**.h** / **.hpp**) în care declarăm clasa (proprietăți, signaturi de constructori, destructori si metode);
- un fișier sursă (**.cpp**) în care furnizăm implementarea clasei noastre.

Numele ambelor fișiere este recomandat sa fie identic cu numele clasei folosind caractere lowercase. Beneficii:

- cod mai lizibil
- când vrem sa cautam o clasa e suficient să căutam fișierul cu numele clasei
- putem consulta setul de funcționalități ale clasei doar aruncând o privire în header.

Exemplu:

fisierul c.h:

```
#ifndef _C_H_
#define _C_H_

class C {
    int x;
    public:
        C (int);
        int get();
        void set(int);
        ~C();
}

#endif //_C_H_
```

fisierul c.cpp:

```
#include <iostream>
#include "c.h"

C::C (int x) {
    this->x = x;
}

int C::get () {
    return this->x;
}

void C::set (int x) {
    this->x = x;
}

C::~C () {
    std::cout << "~C";
}
```

fisierul main.cpp:

```
#include <iostream>
#include "c.h"

int main () {
    C c(3);
    std::cout << c.get() << endl;
    c.set(985);
    std::cout << c.get() << endl;
    return 0;
}
// output:
// 3
// 985
// ~C
```

Separarea declarări de implementare in fisiere are anumite limitari. In cazul claselor template si funcțiilor inline, compilatorul are nevoie ca definirea sa fie completă la momentul includerii header-ului. Prin urmare in aceste situații avem două posibilități:

- implementarea iniline implicită
- implentarea in afara clasei, dar în fișierul header (**.h** / **.hpp**), în loc de fișierul sursă (**.cpp**)

4 Signaturi de clase

Ca in cazul funcțiilor, putem defini o clasa doar prin signatura ei, urmând ca mai târziu sa furnizam implementarea propriu-zisa a clase. De ce am folosi așa ceva? Sunt foarte multe situații in care avem nevoie de o clasa fara a fi nevoie de declararea si implementarea completa a clasei (spre exemplu clase prieten, signaturi de funcții).
Sintaxa pentru a defini o clasa prin signatura este următoarea:

```
class <nume_clasa>;
```

Exemplu:

```
#include <iostream>
#include <string>

using namespace std;

class A;

void showA (A);

int main () {
    A a;
    showA(a);
    return 0;
}

class A {
    int x = 234;
    public:

    int get () {return x;}
};

void showA (A a) {
    cout << "Obiectul este " << a.get() << endl;
}
```

5 Funcții și clase prieten

Funcțiile prieten sunt funcții care au acces la câmpurile private și protected ale unei clase. Pentru a declara o funcție prieten, e suficient să declarăm signatura funcției în momentul declarării clasei atașând cuvântul cheie **friend** în față. Sintaxă:

```
class <nume_clasa> {
    /*
        definitii campuri
    */
    public:
    /*
        definitii metode
    */
    friend <tip_retur> <nume_functie> (<lista_parametri>);
};
```

In cazul in care functia pe care vrem sa fie friend este o metoda a unei clase atunci urmatoarea sintaxa trebuie folosita"

```
    class <nume_clasa_1> {
        /*
            definitii campuri
        */
        public:
        /*
            definitii metode
        */
        friend <tip_retur> <nume_clasa_2>::<nume_functie> (<lista_parametri>);
```

```
    };

```

Putem declara și *clase prieten* pentru o clasă. Ca în cazul funcțiilor, clasele declarate ca prieten au acces la câmpurile private și protected ale clasei declarate. Pentru a declara o clasă prieten trebuie să definim signatura clasei care urmează să fie definită ca prieten. Sintaxă:

```
// signatura clase care urmeaza sa fie declarata ca prieten
class <nume_clasa_1>;

class <nume_clasa_2> {
    /*
        definitii campuri
    */
    public:
        /*
            definitii metode
        */
        friend class <nume_clasa_1>;
};
```

6 Clase imbricate (Nested classes)

In C putem defini structuri de date in interiorul altor structuri de date. Scopul acestei declarării este de defini structura local, deoarece nu ar mai fi nevoie de aceasta structura in alte parți ale programului.
Exemplul:

```
struct Person {
    struct { // structura anonima
        char *street, *city, *county;
    } address;
    char* name;
    unsigned age;
};

int main () {
    struct Person p;
    p.name = "Florin";
    p.age = 39;
    p.address.street = "Academiei Nr.14";
    p.address.county = "Bucharest";
    p.address.city = "Bucharest";

    return 0;
}
```

Același lucru îl putem face si cu clase. Putem declara o clasa in interiorul altei clase. In cazul in care clasa imbricata nu e anonima (ii atașăm un nume), atunci putem defini obiecte de tipul clasei imbricate folosind operatorul rezoluție de scop.

Observatia 1. *Clasele imbricate sunt afectate de specificatorii de acces. O clasa declarata in zona de private nu va putea accesata in exteriorul clasei, pe când o clasa declarata in zona de public va putea fi accesata din zona public.*

```
#include <iostream>
#include <string>

using namespace std;

class Person {
    public:
        class Address {
            string street, city, county;
            public:
                Address () {
                    street = city = county = "";
```

```
                }
                Address (string str, string cty, string cnty) {
                    street = str;
                    city = cty;
                    county = cnty;
                }
        };

    Person (string, unsigned, Address);
    private:
        string name;
        unsigned age;
        Address address;
};

Person::Person(string name, unsigned ag, Address addr) {
    name = name;
    address = addr;
    age = ag;
}

int main () {
    Person::Address a ("Academiei Nr.14", "Bucharest", "Bucharest");
    Person p("Florin", 39, a);
    return 0;
}
```

7 Listă de inițializare

Atunci când implementăm un constructor, imediat după lista de parametri, putem enumera felul în care se inițializează câmpurile clasei. În lista de inițializare putem specifica cum să apelăm constructorul pentru câmpurile alocate static, inițializa câmpurile const sau specifica apeluri explicite ale constructorilor clasei de bază (la moștenire). Lista de inițializare poate fi folosită atât la declarații inline implicite cât și în cazul declarărilor normale. Sintaxa:

```
class <nume_clasa> {
    /*
        definitii campuri
    */
    public:
        /*
            definitii metode
        */
        <nume_clasa> (<lista_parametri>) : <lista_initializare> {
            // implementare constructor
        }
};

<nume_clasa>::<nume_clasa>(<lista_parametri>) : <lista_initializare> {
    // implementare constructor
}
```

8 Metode și câmpuri statice

Câmpurile statice sunt proprietăți care sunt împărțite de toate obiectele unei clase. Un câmp static este inițializat cu valoarea default pentru tipul de date declarat, dacă altă inițializare nu este specificată. Pentru a declara un câmp static trebuie folosit cuvântul cheie *static*. Sintaxă:

```
class <nume_clasa> {
    /*
        definitii campuri
    */
    static <tip_date> <nume_camp>;
    public:
```

```
    /*
        definitii metode
    */
};

// initializare camp static
<tip_date> <nume_clasa>::<nume_camp> = <valoare_initiala>;
```

O clasă poate avea și *metode statice*. Metodele statice nu au pointerul **this** și pot accesa doar câmpuri statice și pot apela doar alte metode statice ale clase. Metodele statice nu pot fi apelate folosind obiect, singura modalitate de apelare fiind folosirea operatorului rezoluție de scop:

```
    <nume_clasa>::<nume_metoda>(parametri);
```

Pentru a declara o metodă statică e sufcient să adăugă cuvântul cheie static în fața signaturii metodei. Sintaxă:

```
class <nume_clasa> {
    /*
        definitii campuri
    */
    public:
    /*
        definitii metode
    */
    static <tip_retur> <nume_metoda>(lista_parametri);
};
```

Exercitii

1. Implementati clasa **List** (lista de numere intregi) care are urmatoarele functionalitati:
   (a) Constructori de copiere si cu parametrii;
   (b) Destructor
   (c) Metoda pentru adaugare unui element in lista
   (d) Metoda pentru eliminarea unui element din lista dupa index/valoare (dupa valoare se elimna prima aparitie intalnita)
   (e) Metoda pentru a returna numarul de elemente din lista
   (f) Metoda pentru a returna elmentul de pe pozitia i
   (g) Metoda pentru a determina elementul maxim din vector.