

Seminarul II

1 Pointerul **this**

În fiecare dintre constructorii, destructorii și metodele clasei avem un pointer care indică către adresa obiectului curent, numit **this**. Scopul acestui pointer este de a putea menționa explicit că ne referim la o metodă sau la o proprietate a obiectului curent.

În C++, folosirea pointerului **this** nu este necesară pentru a putea accesa o proprietate a obiectului curent, compilatorul va deduce din context dacă ne referim sau nu la o proprietate a obiectului. În alte limbaje de programare (e.g. Java, C#) folosirea pointerului **this** este obligatorie.

```
1 class C {
2     int x;
3     public:
4         C(int x) {
5             this->x = x;
6         }
7 };
```

2 Declarații inline

Atunci când declarăm o clasă avem două posibilități pentru a oferi implementarea constructorilor, destructorilor și a metodelor:

- în interiorul declarației clasei;
- în exteriorul declarației clasei;

Implementarea în interiorul declarației clasei se mai numește și implementare **inline**. Acest tip de implementare spune compilatorului că acea zonă de cod trebuie să fie executată cât mai rapid posibil. Compilatorul va încerca să încarce implementarea în memoria cache a procesorului.

Implementarea în exteriorul declarației clasei presupune că fiecare constructor/destructor/metodă trebuie să aibă o semnătură declarată în interiorul clasei:

```
1 [<tip_retur>] <nume_simbol>(<lista_parametrii>);
```

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

```
1 [<tip_retur>] <nume_clasa>::<nume_simbol>(<lista_parametrii>) {
2     /* implementarea propriu-zisă */
3 }
```

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

```
1 inline [<tip_retur>] <nume_clasa>::<nume_simbol>(<lista_parametrii>) {
2     /* implementarea propriu-zisă */
3 }
```

Exemplu:

```
1 class Stack {
2     int *stack;
3     unsigned size , tos;
4
5     public:
6         // constructorul fara parametrii implementat implicit inline
7         Stack () {
8             size = 10;
9             stack = new int [size];
10            tos = -1;
11        }
12
13        // Signatura constructorul cu parametrii ,
14        // signatura metodei de adaugare element in stiva si
15        // signatura destructorul care vor implementata te in afara clasei
16        Stack (int);
17        void push(int);
18        ~Stack();
19 };
20
21 inline Stack::Stack (int n) { // implementare inline explicita
22     size = n;
23     stack = new int [size];
24     tos = -1;
25 }
26
27 void Stack::push (int x) {
28     if (tos == size - 1) {
29         return;
30     }
31     stack[++tos] = x;
32 }
33
34 inline Stack::~Stack () { // implementare inline explicita
35     delete [] stack;
36     size = 0;
37     tos = -1;
38 }
```

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:

fișierul c.h:

```
1 #ifndef _C_H_
2 #define _C_H_
3
4 class C {
5     int x;
6     public:
7         C (int);
8         int get();
9         void set (int);
10        ~C();
11 }
12
13 #endif // _C_H_
```

fișierul c.cpp:

```
1 #include <iostream>
2 #include "c.h"
3
4 C::C (int x) {
5     this->x = x;
6 }
7
8 int C::get () {
9     return this->x;
10 }
11
12 void C::set (int x) {
13     this->x = x;
14 }
15
16 C::~~C () {
17     std::cout << "~C";
18 }
```

fișierul main.cpp:

```
1 #include <iostream>
2 #include "c.h"
3
4 int main () {
5     C c(3);
6     std::cout << c.get() << endl;
7     c.set(985);
8     std::cout << c.get() << endl;
9     return 0;
10 }
11 // output:
12 // 3
13 // 985
14 // ~C
```

Separarea declarații de implementare în fișiere are anumite limitări. În cazul claselor template și funcțiilor inline, compilatorul are nevoie ca definirea să fie completă la momentul includerii header-ului. Prin urmare în aceste situații avem două posibilități:

- implementarea inline implicită
- implementarea în 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:

```
1 class <nume_clasa>;
```

Exemplu:

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 class A;
7
8 void showA (A);
9
10 int main () {
11     A a;
12     showA(a);
13     return 0;
14 }
15
16 class A {
17     int x = 234;
18     public:
19
20     int get () {return x;}
21 };
22
23 void showA (A a) {
24     cout << "Obiectul este " << a.get() << endl;
25 }
```

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:

```
1 class <nume_clasa> {
2     /*
3         definitii campuri
4     */
5     public:
6     /*
7         definitii metode
8     */
9     friend <tip_retur> <nume_functie> (<lista_parametri>);
10 };
```

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

```
1 class <nume_clasa_1> {
2     /*
3         definitii campuri
4     */
5     public:
6     /*
7         definitii metode
8     */
9     friend <tip_retur> <nume_clasa_2>::<nume_functie> (<lista_parametri>);
```

```
10     };
11
```

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 semnatura clasei care urmează să fie definită ca prieten. Sintaxă:

```
1 // semnatura clase care urmeaza sa fie declarata ca prieten
2 class <nume_clasa_1>;
3
4 class <nume_clasa_2> {
5     /*
6         definitii campuri
7     */
8     public:
9         /*
10            definitii metode
11        */
12        friend class <nume_clasa_1>;
13};
```

6 Clase imbricate (Nested classes)

În C++ putem defini structuri de date în interiorul altor structuri de date. Scopul acestei declarații este de a defini structura local, deoarece nu ar mai fi nevoie de această structură în alte părți ale programului.

Exemplu:

```
1 struct Person {
2     struct { // structura anonima
3         char *street, *city, *county;
4     } address;
5     char* name;
6     unsigned age;
7 };
8
9 int main () {
10     struct Person p;
11     p.name = "Florin";
12     p.age = 39;
13     p.address.street = "Academiei Nr.14";
14     p.address.county = "Bucharest";
15     p.address.city = "Bucharest";
16
17     return 0;
18 }
```

Același lucru îl putem face și cu clase. Putem declara o clasă în interiorul altei clase. În cazul în care clasa imbricată nu este anonimă (îi atașăm un nume), atunci putem defini obiecte de tipul clasei imbricate folosind operatorul rezoluție de scop.

Observația 1. Clasele imbricate sunt afectate de specificatorii de acces. O clasă declarată în zona de *private* nu va putea fi accesată în exteriorul clasei, pe când o clasă declarată în zona de *public* va putea fi accesată din zona *public*.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 class Person {
7     public:
8         class Address {
9             string street, city, county;
10            public:
11                Address () {
12                    street = city = county = "";
```

```

13     }
14     Address (string str, string cty, string cnty) {
15         street = str;
16         city = cty;
17         county = cnty;
18     }
19 };
20
21 Person (string, unsigned, Address);
22 private:
23     string name;
24     unsigned age;
25     Address address;
26 };
27
28 Person::Person(string name, unsigned ag, Address addr) {
29     name = name;
30     address = addr;
31     age = ag;
32 }
33
34 int main () {
35     Person::Address a ("Academiei Nr.14", "Bucharest", "Bucharest");
36     Person p("Florin", 39, a);
37     return 0;
38 }

```

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:

```

1 class <nume_clasa> {
2     /*
3         definitii campuri
4     */
5     public:
6         /*
7             definitii metode
8         */
9         <nume_clasa> (<lista_parametri>) : <lista_initializare> {
10             // implementare constructor
11         }
12 };
13
14 <nume_clasa>::<nume_clasa>(<lista_parametri>) : <lista_initializare> {
15     // implementare constructor
16 }

```

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:

```

1 class <nume_clasa> {
2     /*
3         definitii campuri
4     */
5     static <tip_date> <nume_camp>;
6     public:

```

```

7      /*
8         definitii metode
9      */
10 };
11
12 // initializare camp static
13 <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:

```

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

```

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

```

1  class <nume_clasa> {
2      /*
3         definitii campuri
4      */
5      public:
6      /*
7         definitii metode
8      */
9      static <tip_retur> <nume_metoda>(lista_parametri);
10 };

```

Exercitii

1. Implementati clasa **List** (lista de numere intregi) care are urmatoarele functionalitati:

- Constructori de copiere si cu parametri;
- Destructor
- Metoda pentru adaugare unui element in lista
- Metoda pentru eliminarea unui element din lista dupa index/valoare (dupa valoare se elimna prima aparitie intalnita)
- Metoda pentru a returna numarul de elemente din lista
- Metoda pentru a returna elmentul de pe pozitia i
- Metoda pentru a determina elementul maxim din vector.