LABORATORUL 2
Clase. Encapsulare. Constructori. Desctrutori

1 Clase

O clasa reprezinta extensie a structura de date, unde, pe langa membrii care pastreaza starea/ datele (*proprietati*), exista si functii ca membrii (*metode*).

```
class <nume_clasa> {
    <specificator_access>:
        /* definitii proprietati & metode */
    <specificator_access>:
        /* definitii proprietati & metode */

    <numeClasa> (); // constructor
    ~<nume clasa> (); // destructor
} <list_obiecte>;
```

Exemplu:

```
class Point {
    int x = 3, y = -5;

    public:
        void show () {
            std::cout << "(" << x << ", " << ")";
        }
} p;
```

Se pot declara clase folsind si `struct` :

```
struct <nume_structura> {
    <specificator_access>:
        /* definitii proprietati & metode */
    <specificator_access>:
        /* definitii proprietati & metode */

    <nume_structura> (); // constructor
    ~<nume_structura> (); // destructor
} <list_obiecte>;
```

Exemplu:

```
struct Point {
    int x = 3, y = -5;
} p;
```

2 Specificatori de access

Un concept de baza in POO este **encapsularea informatiei**: ascunderea informatiei si oferirea unei interfete prin care se pot executa un set de operatii pe acea informatie.

Q: De ce encapsulare? A: Pentru ca, de cele mai multe ori, operarea directa pe datele unei structuri poate duce la rezulate neprevazute atunci cand modificarea datelor presupune anumiti algoritmi. Prin encapsulare se poate forta ca orice operatie definita pe un set de date sa fie facute doar prin mechanismele definite, care modifica datele intr-o maniera consistenta si sigura.

In C++ avem 3 specificatori:

- `private` - access permis doar in interiorul clasei
- `protected` - access permis in interiorul clasei si in clasele derivate
- `public` - access permis peste tot.

Este recomandat ca poate proprietatile unei clase sa fie declarate cu specificatorul private sau protected

In cazul claselor declarate folsind cuvantul cheie `class` , specificatorul de access default este `private` . In cazul claselor declarare folosing cuvantul cheie `struct` , specificatorul default de access este `public` .

3 Constructori

Un constructor reprezinta "reteta" dupa care se creaza obiectul. Constructorul spune compilatorului cum anume trebuie construit un object dintr-o clasa. Constructorul se declara similar cu metodele, cu urmatoarele precizari: numele este identic cu numele clasei si nu exista tip de retur:

```
1   class Point {
2       int x = 3, y = -5;
3
4       public:
5           Point (int a, int b) {
6               x = a;
7               y = b;
8           }
9           void show () {
10              std::cout << "(" << x << ", " << ")";
11          }
12  };
```

Daca o clasa nu are definit un constructor, compilatorul va atribui un constructor implicit pentru acea clasa.

Exista mai multe tipuri:

- constructor fara parameterii
- constructor cu parameterii
- constructor de copiere

Constructorul cu parametrii si constructorul fara parametrii
sunt apelati la declararea obiectului. Aceste doua tipuri de constructor spun cum anume se creaza un obiect atunci cand nu avem informatii despre obiect sau cand avem informatii despre obiect.

Exemplu:

```
1
2  class Point {
3      int x = 3, y = -5;
4
5      public:
6          Point () {
7              x = 0;
8              y = 0;
9          }
10         Point (int a, int b) {
11             x = a;
12             y = b;
13         }
14 };
15
16 int main () {
17     Point p;      // se apeleaza constructorul fara parameterii
18     P o(1, -10); // se apeleaza constructorul cu parameterii
19     return 0;
20 }
```

Constructorul de copiere este apelat la apelul unei functii care primeste ca parametru un obiect de tipul clasei. Scopul este sa se creeze o copie a obiectului curent, care sa fie folosita in apelul functiei, a.i. dupa terminarea apelului obiectul curent sa ramana nemodificat.

```
1  class Point {
2      int x = 3, y = -5;
3
4      public:
5          Point (Point& p) {
6              x = p.x;
7              y = p.y;
8          }
9  };
10
11 void foo (Point o) {
12     /* corpul functiei */
13 }
14
15 int main () {
16     Point p;
17     Point o(p); // se apeleaza explicit constructorul de copiere
18     f(p);       // se apeleaza implicit constructorul de copiere
19     return 0;
20 }
```

Este absolut necesar ca parameterul pasat catre constructorul de copiere sa fie pasat prin referinta. Altfel se ajunge la un lant infinit de apeluri recursive.

4 Destructori

Destructorul este opusul constructorului. Scopul unui destructor este sa spuna compilatorului cum anume se distruge obiectul. Destructorul este apelat automat de compilator atunci cand durata de viata a unui obiect ajunge la sfarsit.

La fel ca in cazul constructorilor, destructori se declara ca orice alta metoda, avand numele clasei cu ~ in fata si niciun tip de retur

```
class Point {
    int x, y;

    public:
        Point () {
            x = 3;
            y = 44;
        }

        ~Point () {
            std::cout << "Obiect distrus";
       zx }
};

int main () {
    Point p;
    return 0;
} // se apeleaza destructorul

```

Daca nu este declarat un destructor compilatorul creaza un destructor implicit.

5 Exercitii

1. Declarați clasa **Nod** care are următoarele două proprietăți: **info** de tip **int**, informația din nod, și **next** de tip **Nod***, adresa unui alt nod. Implementați constructor (cu parametri, fără parameteri) și destructor pentru clasa **Nod**. Este nevoie de destructor?

2. Implementați pentru clasa **Nod** metodele **setInfo**, care primește ca parametru un număr întreg pe care îl setează în câmpul **info**, și **getInfo**, care întoarce valoarea stocată în câmpul **info**.

3. Implementați pentru clasa **Nod** metodele **setNext**, care primește ca parametru un pointer către **Nod** pe care îl setează în câmpul **next**, și **getNext**, care întoarce valoarea stocată în câmpul **next**.

4. Declarați clasa **Listă** care are următoarele proprietăți: **start**, de tip **Nod*** – pointer către primul element din listă, **end**, de tip **Nod*** – pointer către ultimul elementul din listă, și **size**, de tip **unsigned** – numărul de elemente din listă.

5. Implementați constructorii cu următoarele signaturi:

- **Listă(int x)** `//crează o listă care conține un elementul x`
- **Listă(int x, int y)** `//crează o listă de x elemente, fiecare cu valoarea y`

6. Implementați următoarele metode:

- **void insert(int x); //inserează valoarea `x` la sfârșitul listei**
- **void insertAt(int x, int i); //inserează elementul `x` pe poziția `i`, dacă `i` este mai mare ca `size` se inserează la sfârșitul listei**
- **int get(int i); //întoarce elementul de pe poziția `i`, MAX_INT dacă `i` nu este un indice valid**
- **int length(); //întoarce numărul de elemente din listă**
- **void remove(int i); //șterge elementul de pe poziția i**

7. Implementați functia **void f (Listă l, int x)** care afișează lista **l** și insearează la sfârșitul listei valoarea **x**. Apelați de două sau mai multe ori această funcție folosind același obiect **l**. Ce observați?

8. Rezolvați problema descoperită a la punctul anterior.

9. Implementați destructorul pentru clasa **Listă**.

10. Implementați metoda **Listă reverse()** care întoarece un nou obiect de tip **Listă** care conține elementele listei curente în ordine inversă.

11. Implementați metodele **void removeFirst()** și **void removeLast()** care șterg primul, respectiv ultimul, element din listă.

12. Implementați metoda **bool hasDuplicates()** care întoarce **true** dacă lista conține elemente duplicate, `false` altfel.

13. Implementați metoda **bool has(int x)** care întoarce **true** dacă lista conține elementul **x**, **false** altfel.

14. Implementați metoda **bool isEmpty()** care întoarce **true** dacă lista nu conține elemente **false** altfel.