LABORATORUL 5

1 Moștenire

Suntem deseori în situația de a avea nevoie de a avea mai multe tipuri de date care au în comun mai multe caracteristici sau , o dată cu dezvoltarea aplicației noastre, sa fim nevoiți sa exitindem un tip de date existent fără sa pierdem forma curenta. O soluție evidentă (non-POO) ar fi sa cream noul tip plecând de la o copie a tipului curent. Genul acesta de soluție nu este foarte practică. Duplicarea codului, pe lângă faptul ca va crește numărul de linii de cod din proiect, duce la o durată mai mare a procesului de compilare. Mai mult o problema identificată în codul comun trebuie fixată separat în fiecare copie a codului.

În programarea orientată pe obiecte putem crea tipuri noi de date plecând da la un tip de date, fără a fi nevoie să duplicăm cod. Această funcționalitate se numește *moștenire*. Astfel putem crea funcționalități noi fără să pierdem funcționalități vechi și fără să creștem dimensiunea proiectului într-o manieră necontrolată (refolosim cod).

În C++ (dar nu numai) putem moșteni unul sau mai multe tipuri de date în crearea tipului nou de date. Clasa din care se moștenește se numește clasă de bază, iar clasa care moștenește se numește clasă derivată. Sintaxă moștenire:

```
class <nume> : <spec_acces_1> <baza_1>, ... , <spec_acces_n> <baza_n> {
    // definitie clasa
};
```

Specificatorii de access au rolul de specifica ce tip de access vor avea proprietățile moștenite în clasa derivată. Următorul tabel descrie cum specificatorul de access folosit la moștenire influențează specificatorul final de acces al unei prorietăți moștenite:

Moștenire \ bază	**public**	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	inaccesibil	inaccesibil	inaccesibil

Când instanțiem un obiect al unei clase derivate, înainte de a se executa contrutorul clasei, se va apela implicit, de catre compilator, contructorul fără parameterii al clasei de bază. Acest comportament poate fi modificat prin folosirea listei de inițializare în constructorul clasei derivate și apelarea explicită a constructorului clasei de bază dorit.

Când este distrus un obiect, destructorii sunt apelați în ordine inversă constructorilor: mai întâi destructorul clase de derivate, apoi constructorul clase de bază.

```
#include <iostream>

using namespace std;

class A {
    public:
    A () {cout << "A";};
    A (int x) {cout << "A" << x;}
    ~A () {cout << "~A";};
};

class B: public A {
    public:
    B () : A(3) {cout << "B";}
    ~B () {cout << "~B";}
};

class C: public B {
    public:
    C () {cout << "C";}
    ~C () {cout << "~C";}
};

```

```
int main () {
    C c;
    return 0;
}

// afiseaza A3BC~C~B~A
```

Câmpurile și metodele statice sunt și moștenite în clasa derivată. Cămpurile statice sunt partajate cu clasa de bază: o modificare a unui câmp static în clasa de bază va fi vazută și în clasa derivată și viceversa.

Dacă implementarea metodelor moștenite nu este potrivită pentru clasa derivată, atunci putem să suprascriem metoda și să îi oferim o nouă implementare.

Atunci când lucram cu pointeri și referințe către tipul de bază, putem asigna în acestea obiecte de tip derivat. Acest procedeu se numește upcasting și este posibil deoarece clasa derivat o parte comună cu clasa de bază. Aceeași afirmație nu este adevarată și în sens invers doarece nu întotdeauna tipul de bază conține exact aceleași proprietăți ca tipul derivat (există downcasting însă nu e mereu posibil).

```
#include <iostream>

using namespace std;

class A {
public:
    static void f () {
        cout << "A" << endl;
    }
};

int A::a = 0;

class B: public A {
public:
    static void f () {
        cout << "B" << endl;
    }
};


int main () {
    A *a = new B();
    a->f(); // se apeleaza A::f()
    return 0;
}
```

Dacă moștenim multiple clase, este posibil să avem două proprietăți cu același nume. Pentru a putea accesa proprietățiele e nevoie sa specificăm numele clasei de bază din care vrem sa accesăm proprietatea (dacă nu facem asta compilatorul va spune ca simbolul accesat este ambiguu):

```
#include <iostream>

using namespace std;

class A {
    protected:
        int x;
};

class B {
    protected:
        int x;
};

class C: public A, public B {
    public:
        C () {
            A::x = 3;
        }
        friend void showC (C c);
};

void showC (C c) {
    cout << c.A::x << " ";
}

int main () {
    C c;
```

```
29      showC(c);
30      return 0;
31  }
```

Exerciții

Implementați o aplicație de gestionare a evenimentelor. Fiecare eveniment are un nume, o locație și o dată la care are loc (minut, ora zi, luna, an) și o durată. Evenimentele pot avea loc o singură data sau pot fi recurente (zilnic, saptamanal, lunar, anual) și în funcție de numarul de persoane implicat, evenimentele pot fi individuale sau de grup (o listă de persoane implicate în eveniment este oferită).Fiecare eveniment are un ID unic de forma $EV-X-nr\_ord$, unde nr_ord este numarul de ordine al evenimentului, iar X are valoare I pentru evenimente individuale, R pentru evenimente recurente, IR pentru evenimente individuale recurente si GR pentru evenimente de grup recurente. Aplicația trebuie sa permită crearea de noi evenimente, listarea evenimentelor și stergearea unui eveniment după ID.