Exerciții moștenire

1. Să presupunem că în cadrul unui program am identificat nevoia de a reține „denumirea” pentru diferite tipuri de obiecte din clase diferite, care nu au vreo legătură între ele. Putem scrie mai puțin cod și putem refolosi aceeași definiție a acestui câmp folosind conceptul de **moștenire**.

   Definiți o clasă denumită **`CuNume`** care să permită claselor care o moștenesc să primească un „nume”[1]. Cu alte cuvinte, această clasă ar trebui să aibă o dată membru **privată** denumită `nume`, de tip `string`, care va reține „eticheta” obiectului. Definiți un constructor parametrizat pentru aceasta, care va inițializa câmpul respectiv (constructorul poate să fie `public` sau `protected`). Implementați un *getter* și un *setter* pentru acest câmp, ca metode publice.

   Această clasă nu are o utilitate de sine stătătoare (nu conține vreo logică în plus față de o simplă variabilă de tip `string`), dar oferă un mod facil de a adăuga funcționalitatea de „etichetare” la clasele noastre.

   Definiți trei clase care să **moștenească** (public) din clasa `CuNume`, de exemplu `Persoana`, `Produs` și `AnimalDeCompanie` (puteți adăuga și alte câmpuri private care vi se par relevante la ele). Pentru fiecare, definiți un **constructor** care să primească cel puțin denumirea noului obiect ca parametru și să o transmită la constructorul clasei de bază. Creați câte un obiect din fiecare dintre aceste clase și verificați că puteți apela setter-ul/getter-ul pentru proprietatea `nume`.

2. Un dezavantaj al modului în care am definit mixin-ul `CuNume` la exercițiul anterior ar putea fi că la toate putem face *upcasting* la clasa de bază, deși am vrea să fie doar un detaliu de implementare.

   De exemplu, următorul cod funcționează:

```
1  Persoana persoana;
2  Produs produs;
3  AnimalDeCompanie animal;
4
5  CuNume* variabila = &persoana;
6  variabila->set_nume("Ionel");
7
8  variabila = &produs;
9  variabila->set_nume("Ananas");
```

[1] O clasă de acest tip poartă denumirea de *mixin*. Acest *pattern* este popular și în alte limbaje orientate pe obiect, cum ar fi Python.

```
10
11 variabila = &animal;
12 variabila->set_nume("Rex");
```

Am vrea ca toate aceste clase să aibă un câmp pentru a reține denumirea lor, dar nu am vrea să fie considerate în vreun fel „înrudite”. Putem obține acest rezultat schimbând tipul de moștenire din **public** în **privat**.

Înlocuiți peste tot moștenirile `public CuNume` cu `private CuNume`, iar apoi adăugați declarațiile

```
1 using CuNume::<denumire getter>;
2 using CuNume::<denumire setter>;
```

în secțiunea de membrii `public` din fiecare clasă care îl moștenește.

Testați că încă funcționează codul de la exercițiul anterior, dar cel cu *upcasting* din acest exercițiu nu mai este acceptat de compilator.

3. În momentul în care moștenim o clasă și definim o nouă metodă cu aceeași signatură, metoda din clasa de bază este implicit ascunsă dar putem să o apelăm în continuare dacă vrem acest lucru.

   Fiind dată următoarea ierarhie de moștenire:

```
1     class A {
2     public:
3         void f() {
4             std::cout << "f din A" << std::endl;
5         }
6     };
7
8     class B : public A {
9     public:
10        void f() {
11            std::cout << "f din B" << std::endl;
12        }
13    };
14
15    class C : public B {
16    public:
17        void f() {
18            std::cout << "f din C" << std::endl;
19        }
20    };
```

Creați un obiect de tip `C` iar apoi apelați pe el metoda `f` din `C`, metoda `f` din `B` și metoda `f` din `A`, folosind *qualified-id*-ul metodei.

4. În cadrul acestui exercițiu, vă veți familiariza cu utilizarea moștenirii și a **polimorfismului la execuție** pentru a schimba dinamic implementarea folosită pentru o metodă, în funcție de tipul obiectului pe care o apelați.

   Ați mai întâlnit deja acest comportament în momentul în care ați supraîncărcat operatorii `<<` și `>>`: primeați un parametru de tip `ostream&`/`istream&` și îl foloseați ca să afișați/citiți datele membre din clasa voastră. Ulterior, puteați apela acești operatori supraîncărcați cu niște instanțe concrete de stream-uri: `cout`/`cin`, `ofstream`/`ifstream` etc.

   Începeți prin a defini o „interfață”[2] (o clasă care nu are date membru, doar metode pur virtuale și destructorul virtual) numită `Shape`, care să reprezinte o figură geometrică. Această interfață va conține următoarele două metode publice:

```
virtual double compute_perimeter() const = 0;
virtual double compute_area() const = 0;
```

   Definiți următoarele clase, care să implementeze (să **moștenească**) interfața de mai sus:

   - Clasa `Triangle`, care reține baza și înălțimea unui triunghi.
   - Clasa `Rectangle`, care reține lățimea și lungimea unui dreptunghi.
   - Clasa `Circle`, care reține raza unui cerc.

   Definiți constructori pentru fiecare dintre aceste clase și suprascrieți metodele `compute_perimeter` și `compute_area` pentru acestea[3], care vor calcula perimetrul, respectiv aria fiecărei figuri geometrice.

   Definiți subprogramul `print_shape_size` în felul următor:

```
void print_shape_size(Shape& shape)
{
    std::cout << "Figura geometrica are perimetrul "
        << shape.compute_perimeter()
        << " si aria "
```

[2] Dacă în C++ o *interfață* este doar o clasă obișnuită căruia îi impunem anumite constrângeri, în limbaje ca Java sau C# există un cuvânt cheie special pentru definirea interfețelor.

[3] **Atenție:** pentru a suprascrie metodele din clasa de bază, cele din clasele moștenitoare trebuie să aibă exact aceeași **signatură** (denumire, tip de date returnat, parametrii și modificatorul `const`, dacă e cazul).

```
6           << shape.compute_area()
7           << std::endl;
8   }
```

În programul principal, definiți câte o variabilă de tip `Triangle`, `Rectangle`, respectiv `Circle`.

Apelați subprogramul `print_shape_size` pe rând cu fiecare dintre aceste variabile.

Aici avem un exemplu de *polimorfism la execuție*: deși (formal) apelăm mereu aceleași metode (`compute_perimeter` și `compute_area`) pe același tip de date (parametrul de tip `Shape&`), ajung să se execute funcții diferite.

5. Acest exercițiu vă arată cum putem gestiona **moștenirea în diamant** în C++.

   Începeți prin a defini clasele de care vom avea nevoie:

   - Definiți o clasă `Product`, care va reprezenta un produs dintr-un supermarket. Aceasta reține prețul de bază al unui produs (o variabilă membru `double price`, cu modificatorul de acces `protected`) și care are o metodă publică *virtuală* denumită `get_price`, care returnează direct prețul de bază.

     Observație: această clasă ar trebui să aibă definit și destructorul ca fiind `virtual`, deoarece va trebui să alocăm dinamic și să ștergem obiecte din clasa `Product` și din clasele care o moștenesc, dar la care ne vom referi printr-un pointer la clasa de bază `Product`.

   - Definiți clasa `PerishableProduct`, care moștenește `public virtual` clasa `Product` și reprezintă un produs perisabil. Aceasta ar trebui să rețină o variabilă de tip `bool` care indică dacă produsul este aproape de data expirării sau nu. Adăugați și un setter pentru acest câmp. Clasa ar trebui să suprascrie metoda `get_price`, ca să returneze prețul de bază redus cu 10% dacă indicăm că produsul este aproape de data expirării.

   - Definiți clasa `ProductOnSale`, care moștenește `public virtual` clasa `Product` și reprezintă un produs la care s-a aplicat o reducere de preț configurabilă. Aceasta ar trebui să rețină o variabilă `double`, care este o reducere procentuală configurabilă, modificabilă printr-un setter. Clasa ar trebui să suprascrie metoda `get_price`, ca să returneze prețul de bază redus cu reducerea procentuală stocată.

   - Definiți clasa `PerishableProductOnSale`, care moștenește ambele clase de mai sus și le combină funcționalitățile (va calcula procentul

total cu care trebuie redus prețul de bază și îl va aplica în `get_price`).

Vom folosi clasele în felul următor în `main`:

- Creați un vector de obiecte de tip `Product` (sau derivate ale acestuia), alocate dinamic (i.e. un `vector<Product*>`).
- Adăugați în acesta câte un obiect (alocat dinamic) din fiecare dintre clasele `Product`, `PerishableProduct`, `ProductOnSale` și `PerishableProductOnSale`.
- Scrieți un singur `for` care să parcurgă vectorul și să afișeze pentru fiecare produs prețul final calculat de metoda `get_price`.
- La final, nu uitați să ștergeți (să apelați `delete`) pentru fiecare dintre obiectele alocate dinamic din vector.