Exerciții – Partea a 2-a

1. În prima parte a laboratorului, ați avut de implementat o clasă `String` pentru gestionarea facilă a șirurilor de caractere alocate dinamic. Biblioteca standard C++ oferă clasa `std::string` în acest scop, care include mare parte din funcționalitățile pe care le-ați implementat și încă câteva în plus.

   Referințe utile: referință pentru tipul de date `std::string`, utilizarea clasei `std::string` în C++.

   Scopul acestui exercițiu este să vă familiarizeze cu utilizarea `string`. Dacă aveți nevoie să gestionați șiruri de caractere la colocviu, sau dacă ajungeți să lucrați pe C++, este mult mai convenabil și eficient să utilizați acest tip de date interoperabil oferit de limbaj.

   Observație: pentru a putea folosi clasa `string` din biblioteca standard, va trebui să includeți fișierul header `<string>`.

   - Creați o nouă variabilă de tip `string`, inițializată cu constructorul fără parametrii; aceasta va reține șirul vid.

     Referințe utile: constructorii clasei `string`.

   - Creați un nou `string` inițializat cu valoarea unui *string literal* (e.g. `string sir = "acesta este un exemplu"`).

     Observație: *nu* este nevoie să ștergeți explicit memoria alocată pentru un `string`; clasa are definit un destructor care se ocupă de acest lucru.

   - Copiați un `string` într-un alt `string` folosind constructorul de copiere, respectiv operatorul =.

   - Ștergeți conținutul unui `string` folosind metoda `clear`. Verificați că șirul este vid după apelarea acesteia folosind metoda `empty`.

     Referințe utile: metoda `clear`, metoda `empty`.

   - Afișați în consolă șirurile create până acum folosind operatorul <<, care este deja supraîncărcat de către biblioteca standard pentru clasa `string`.

     Referințe utile: operatorul << pentru `std::string`.

   - Citiți un `string` de la tastatură folosind operatorul >> (care este deja supraîncărcat de către biblioteca standard). Acesta permite citirea unui șir de caractere doar până la primul spațiu sau rând nou (până la primul caracter *whitespace*).

Observație: operatorul de citire >> pentru `string` alocă automat câtă memorie este necesară pentru a reține șirul citit, suprascriind orice valoare avea înainte `string`-ul.

Referințe utile: operatorul >> pentru `std::string`.

- Citiți un întreg rând de la tastatură într-un `string` folosind funcția liberă `getline` din biblioteca standard.

  Observație: s-ar putea să fie nevoie să apelați `cin >> ws` (ca să consumați caracterul de rând nou rămas pe linia precedentă) înainte de a apela `getline`, dacă anterior ați citit un `string` cu operatorul >>.

  Observație: funcția `getline` pentru `string` alocă automat câtă memorie este necesară pentru a reține rândul citit, suprascriind orice valoare avea înainte `string`-ul.

  Referințe utile: manipulatorul `std::ws`, funcția `std::getline` (cu parametru `string`).

- Determinați lungimea unui șir de caractere citit de voi de la tastatură folosind metoda `length`. Alternativ, puteți folosi metoda `size` (sunt sinonime).

  Referințe utile: metoda `length`, metoda `size`.

- Creați un șir de caractere nevid și accesați caracterul de pe poziția i (aleasă de voi) folosind operatorul de indexare `[]` (care este deja supraîncărcat de către biblioteca standard).

  Ce se întâmplă dacă încercați să accesați un caracter din afara șirului?

  Referințe utile: operatorul `[]`.

- Faceți același lucru ca la subpunctul precedent, dar de data aceasta folosiți metoda `at` ca să accesați caracterul de pe poziția i.

  Ce se întâmplă de data aceasta dacă încercați să accesați un caracter din afara șirului?

  Referințe utile: metoda `at`.

- Clasele container din biblioteca standard C++ permit în general iterarea prin elemente folosind *iterator pattern* (i.e. o clasă ajutătoare care reține elementul la care ne aflăm, permite accesarea/modificarea acestuia și trecerea la elementul următor/anterior).

Pentru a itera prin toate caracterele unui `string`, puteți folosi o secvență de cod similară cu următoarea:

```
for (string::iterator it = sir.begin();
     it != sir.end();
     ++it)
{
    char caracter = *it;
    std::cout << caracter << std::endl;
}
```

Alternativ, în C++11 se poate evita menționarea explicită a tipului de date al iteratorului folosind cuvântul cheie `auto`:

```
for (auto it = sir.begin();
     it != sir.end();
     ++it)
{
    char caracter = *it;
    std::cout << caracter << std::endl;
}
```

Mai succint, tot începând cu C++11 puteți folosi un *range-for*:

```
for (char caracter : sir) {
    std::cout << caracter << std::endl;
}
```

Dezavantajul este că în acest mod nu mai aveți la fel de mult control asupra iterării (de exemplu, nu puteți sări caractere).

Referințe utile: utilizarea iteratorilor în biblioteca standard C++, tipuri de iteratori, range-based `for` statement.

- Definiți un `string` inițializat cu un șir de caractere care reprezintă un număr întreg (e.g. `"123"`). Convertiți-l într-un `int` folosind funcția `std::stoi`.

  Faceți același lucru cu un șir de caractere care reprezintă un număr întreg *reprezentat în binar* (e.g. `"10010"`) și convertiți-l într-un `int`, de data aceasta dând valoarea 2 pentru parametrul `base` al funcției `std::stoi` (pentru `"10010"`, ar trebui să obțineți numărul întreg 18).

  Referințe utile: funcțiile `std::stoi`, `std::stol`, `std::stoll`.

- Citiți de la tastatură un enunț scris pe un singur rând (folosind funcția `std::getline`) și apoi un cuvânt (folosind operatorul >>). Folosiți metoda `find` a clasei `string` ca să vedeți dacă cuvântul respectiv se găsește în enunț, respectiv care este prima poziție pe care se găsește.

  Observație: metoda `find` returnează un număr întreg fără semn, care pentru corectitudine ar trebui păstrat într-o variabilă de tipul `size_t` (care pe majoritatea sistemelor va fi un alias pentru `unsigned long long` sau similar).

  Observație: metoda `find` va returna constanta `std::string::npos` dacă nu a reușit să găsească subșirul respectiv în șirul de caractere pe care a fost apelată.

  Referințe utile: metoda `find`, constanta `std::string::npos`, tipul de date `std::size_t`.

2. În prima parte a laboratorului, ați avut de implementat o clasă `IntVector` care gestiona un vector de numere întregi, alocat dinamic. Ar fi un efort de mentenanță foarte mare pentru biblioteca standard să definească clase pentru fiecare tip de date în parte (ar veni `IntVector`, `ShortVector` etc.), codul din implementările lor ar fi foarte similar, iar acestea nu ar putea fi folosite oricum pentru a gestiona vectori de tipuri de date definite de dezvoltator.

   Soluția este ca biblioteca standard să furnizeze o clasă șablon (*template class*), pe care o putem instanția cu orice tip de date vrem, fie el *built-in* sau definit de noi[1]. Veți învăța cum funcționează precis și cum pot fi definite clasele/funcțiile șablon mai târziu, dar pentru început este suficient să știți că puteți crea o variabilă de tip vector ca în următoarele exemple:

```
1 std::vector<int> vector_de_intregi;
2 std::vector<std::string> vector_de_siruri_de_caractere;
3 std::vector<MyClass> vector_de_obiecte_din_clasa_mea;
4 std::vector<std::vector<SomeClass*>>
5     vector_de_vectori_de_pointeri;
```

   (puteți să nu mai puneți `std::` în față dacă aveți `using namespace std`)

   Observație: pentru a putea folosi clasa șablon `vector` din biblioteca standard, va trebui să includeți fișierul header `<vector>`.

   - Creați un vector gol de numere de tip `double`, folosind constructorul fără parametri.

[1] Cu mențiunea că acest tip de date trebuie să respecte anumite constrângeri, de exemplu să aibă un constructor fără parametri și constructor/operator = de copiere, toate accesibile public.

Referințe utile: constructorii clasei `vector`.

- Creați un vector de numere de tip `double`, care să conțină numărul 2.5 repetat de 10 ori (folosiți *fill* constructorul clasei `vector`).
- Creați un vector plecând de la un șir de numere `double` fixat de voi în cod (e.g. `{ 2.5, 0, 3.1, -4.3, 1 }`) (folosiți *range* constructorul).
- Copiați un vector creat de voi într-un alt vector de același tip. Observați că biblioteca standard definește constructorul de copiere și operatorul = pentru noi; aceștia alocă memorie pentru noul vector și copiază pe rând fiecare element.
- Citiți un număr natural *n* de la tastatură, redimensionați vectorul ca să conțină *n* numere de tip `double` folosind metoda `resize`, iar apoi citiți fiecare număr în vector (folosiți operatorul `[]`, care este supraîncărcat pentru clasa `vector`).

  Referințe utile: metoda `resize`, operatorul `[]`.
- Parcurgeți și afișați elementele vectorului citit anterior, *în ordine inversă*. Folosiți iteratori ca la exercițiul cu `string`, dar de data aceasta plecând de la `rbegin` și mergând până la `rend` (va trebui să declarați variabila din `for` ca `std::vector<double>::reverse_iterator`, deoarece metodele `rbegin`/`rend` returnează un alt tip de clasă ajutătoare față de `begin`/`end`).

  Referințe utile: `reverse_iterator`, metoda `rbegin`, metoda `rend`.
- Adăugați un număr nou la final folosind metoda `push_back` (aceasta va crește automat capacitatea alocată a vectorului, dacă este cazul, ca să încapă și elementul adăugat).

  Referințe utile: metoda `push_back`.
- Calculați media aritmetică a unui vector de numere de tip `double`. În acest scop, determinați la execuție câte elemente conține vectorul, prin metoda `size`.

  Referințe utile: metoda `size`.

3. 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:

```
1 virtual double compute_perimeter() const = 0;
2 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:

```
1 void print_shape_size(Shape& shape)
2 {
3     std::cout << "Figura geometrica are perimetrul "
4         << shape.compute_perimeter()
5         << " si aria "
6         << shape.compute_area()
7         << std::endl;
8 }
```

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

[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).

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.

4. 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 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 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.

5. Definiți o clasă de tip *mixin* denumită **`Labelable`** care să permită claselor care o moștenesc să fie „etichetate". Cu alte cuvinte, această clasă ar trebui să aibă o dată membru privată denumită `label`, de tip `string`, care va reține „eticheta" obiectului. 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.

   Referințe utile: definiția conceptului de *mixin*, exemplu de cum se folosește pattern-ul *mixin* în Python.

6. Adăugați mixin-ul `Labelable` la clasele `Shape` și `Product` definite în exercițiile anterioare (folosind moștenirea multiplă). Încercați să setați și să afișați etichetele pentru obiecte de tip `Circle`, respectiv `PerishableProduct`.

7. Supraîncărcați operatorul de indexare `[]` pentru una dintre clasele pe care le-ați implementat la laboratoarele precedente (e.g. `IntVector`, `String`).

   Acesta ar trebui să permită **accesarea** valorii elementului de pe poziția *i*, dar și **modificarea** acestuia (dacă obiectul nu este constant). Mai mult, ar trebui **să arunce o excepție** de tipul `std::out_of_range` dacă parametrul *i* este negativ sau mai mare decât lungimea containerului.

   Observație: putem acomoda cele două situații supraîncărcând de două ori operatorul; o dată pentru când obiectul implicit este mutabil, o dată pentru când acesta este constant. Cele două antete ar trebui să fie (de exemplu, pentru `IntVector`):

```
1 int& operator[](int i);
2 int operator[](int i) const;
```

   În primul caz, returnăm un `int&` ca să putem modifica elementul prin intermediul valorii returnate (să putem scrie, de exemplu, `v[i] = 3`). În al doilea caz, când vectorul de întregi este constant, putem la fel de bine să

returnăm direct un `int`; nu obținem o performanță mai bună returnând prin referință.

Referințe utile: supraîncărcarea operatorului de indexare, aruncarea excepțiilor în C++.

8. Implementați o clasă care să rețină de câte ori a fost instanțiată sau ștearsă pe parcursul execuției unui program.

   Puteți obține această funcționalitate prin definirea unei date membru statice, de tip întreg, care pornește de la 0 și pe care o veți incrementa de fiecare dată în constructor (în cel fără parametrii, dar și cel de copiere) și în operatorul =, respectiv pe care o veți decrementa în destructor.

   Adăugați o metodă statică care să returneze valoarea curentă acestui contor, ca să-l puteți afișa.

   Referințe utile: membrii statici ai unei clase în C++.

9. Implementați o clasă *singleton*, o clasă din care să poată exista o singură instanță pe toată durata de execuție a programului.

   Puteți face acest lucru în doi pași:

   - Marcați ca privat constructorul de inițializare al clasei, respectiv constructorul de copiere și operatorul = (astfel, clasa nu va putea fi instanțiată sau copiată de codul din afara ei).
   - Definiți o metodă statică care să permită obținerea unei referințe la singura instanță a clasei. Aceasta ar trebui să conțină o variabilă locală statică, de tipul clasei respective, pe care să o returneze prin referință.

   Referințe utile: design pattern-ul singleton, implementarea unei clase de tip singleton în C++.