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: interfața tipului de date `std::string`, utilizarea clasei `std::string`.

   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.

   - Creați o nouă variabilă de tip `string`, inițializată cu constructorul fără parametrii; aceasta va reține șirul vid.
   - Creați un nou `string` inițializat cu valoarea unui *string literal* (e.g. `string sir = "acesta este un exemplu"`).
   - 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`.
   - 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*).
   - Citiți un întreg rând de la tastatură într-un `string` folosind funcția liberă `getline` din biblioteca standard.

     Referințe utile: funcția `std::getline`.
   - Afișați în consolă șirurile create până acum folosind operatorul << (care este deja supraîncărcat de către biblioteca standard).
   - Determinați lungimea unui șir de caractere citit de voi de la tastatură folosind metoda `length`. Alternativ, puteți folosi metoda `size` (fac fix același lucru).

     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?

- 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:

```
1 for (string::iterator it = sir.begin();
2       it != sir.end();
3       ++it)
4 {
5     std::cout <<
6 }
```

  Alternativ, în C++11, folosind tipul de date auto:

  Alternativ, necesită tot C++11, folosind range-for:

```
1 for (char c : sir) {
2     std::cout << c << std::endl;
3 }
```

- 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 `IntVector`, `ShortVector` etc. pentru fiecare tip de date în parte, 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 noi.

   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].

3. Definiți o *interfață* (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 avea următoarele două metode publice:

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

   Definiți următoarele clase, care să extindă 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.

[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.

Definiți constructori pentru fiecare dintre aceste clase și suprascrieți metodele `compute_perimeter` și `compute_area` pentru acestea, 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 "
        << shape.compute_area()
        << std::endl;
}
```

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.

4. Supraîncărcați operatorul de indexare `[]` pentru clasa `IntVector` sau `String` din laboratoarele precedente.

   Acesta va permite **accesarea** valorii numărului întreg (respectiv a caracterului) de pe poziția i, dar și **modificarea** acestuia (dacă vectorul nu este constant). Mai mult, operatorul de indexare va arunca o **excepție** de tipul `std::out_of_range` dacă parametrul i este negativ sau mai mare decât lungimea vectorului/șirului.

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

```
int& operator[](int i);
const 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++.

5. Implementați o clasă care să rețină de câte ori a fost instanțiată și de-instanțiată.

   Puteți face asta 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), respectiv decrementa în destructor.

   Adăugați o metodă statică care să returneze valoarea curentă a contorului.

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