Exerciții — Concepte de bază C++

Aceste exerciții vă introduc pe rând în diferite concepte ale limbajului C++ și ale programării orientate pe obiecte. Dificultatea lor crește treptat, unele dintre exercițiile ulterioare făcând referire la sau utilizând noțiuni abordate anterior.

Nu este obligatoriu să le lucrați pe toate, obiectivul principal al laboratorului fiind implementarea temelor de proiect, dar v-ar putea fi utile pentru a vă obișnui cu limbajul și pentru a vă dezvolta abilitățile.

1. Scrieți un program în C++ care să citească de la tastatură două numere întregi a și b și să afișeze suma lor, $a + b$.

   Referințe utile: structura unui program C++, declararea de variabile, operații de citire și de afișare.

2. Scrieți un program în C++ care să citească de la tastatură un număr natural n și apoi să afișeze, pe câte un rând, toate numerele naturale de la 1 la n, cu mențiunea că numerele care sunt divizibile cu 3 vor fi înlocuite cu textul `Fizz`, numerele care sunt divizibile cu 5 vor fi înlocuite cu textul `Buzz`, iar numerele care sunt divizibile și cu 3 și cu 5 vor fi înlocuite cu textul `FizzBuzz`[1].

   Referințe utile: `for` loop, structură condițională `if...else`, operatorul modulo (%).

3. Scrieți un program în C++ care să citească de la tastatură un număr întreg n, care să fie *natural*, *nenul* și *par*.

   Programul îi va cere utilizatorului să introducă la tastatură numărul, va citi un număr întreg și va continua să facă acest lucru până când numărul citit respectă cerințele impuse (să fie > 0 și divizibil cu 2).

   La final, afișați pe ecran valoarea lui $n/2$.

   Observație: puteți folosi acest pattern când aveți nevoie să citiți date de la tastatură care să respecte un anumit format.

   Referințe utile: structură repetitivă `do...while`

4. Limbajul C++ permite *supraîncărcarea* funcțiilor: definirea mai multor subprograme care au *aceeași denumire*, dar diferă prin *tipul* sau *numărul* parametrilor pe care îi acceptă.

[1] Aceasta este celebra problemă *fizz buzz*, despre care se spunea că e o problemă tipică de interviu de angajare ca programator.

Implementați mai multe funcții, toate denumite `absolute_value`, care să calculeze valoarea absolută a unui:

- Număr întreg (un parametru de tip `int`);
- Număr rațional (un parametru de tip `double`);
- Număr complex (doi parametrii de tip `double`, care reprezintă partea lui reală și partea lui imaginară)
- Vector din $\mathbb{R}^3$ (trei parametrii de tip `double`, care reprezintă coordonatele vectorului).

Antetele lor ar trebui să fie:

```
1 double absolute_value(int nr);
2 double absolute_value(double nr);
3 double absolute_value(double real, double imag);
4 double absolute_value(double x, double y, double z);
```

Apelați fiecare dintre aceste funcții din subprogramul principal. Observați cum compilatorul este capabil să determine automat la care variantă a funcției vă referiți.

Referințe utile: supraîncărcarea funcțiilor în C++, `std::sqrt`;

5. Implementați o funcție în C++ care să interschimbe valorile a doi parametri, numere întregi, **transmiși prin referință**.

   Antetul funcției ar trebui să fie: `void interschimba(int& a, int& b);`

   Scrieți și un cod corespunzător în subprogramul principal, care să apeleze acest subprogram și să verifice că funcționează cum trebuie.

   Referințe utile: interschimbarea a două numere întregi, referințe în C++.

6. Implementați o funcție în C++ care să interschimbe valorile către care trimit cei doi parametri de tip pointer la numere întregi (`int*`).

   Pentru a obține un pointer care să trimită la o variabilă deja declarată, putem folosi operatorul „address-of” (ampersandul `&`). Pentru a citi valoarea din zona de memorie indicată de către un pointer (presupus valid), putem folosi operatorul de dereferențiere (asteriscul `*`).

   Antetul funcției ar trebui să fie: `void interschimba(int* a, int* b);`

   O puteți apela în `main` în felul următor:

```
int x = 1, y = 2;
interschimba(&x, &y);
// x ar trebui sa aiba acum valoarea 2,
// iar y valoarea 1
```

Observație: în limbajul C nu există referințe. În acel caz, pointerii sunt singurul mod prin care ne putem referi cu mai multe nume la aceeași zonă de memorie.

Referințe utile: utilizarea pointerilor în C++, operatorii de referențiere și de dereferențiere, intereschimbarea a două numere folosind pointeri.

7. Definiți o nouă structură `Complex` (folosiți cuvântul cheie `struct`), care să rețină **partea reală** și **partea imaginară** a unui număr complex (numere reale stocate ca `double`).

   Referințe utile: cum se declară o structură în C++.

   Implementați următoarele funcționalități pentru aceasta:

   - Un constructor fără parametrii, care să instanțieze numărul complex 0.

     Referințe utile: constructori.

   - Un constructor care primește ca parametrii partea reală și partea imaginară a unui număr complex și le salvează în variabilele membru corespunzătoare. Folosiți o listă de inițializare în constructor pentru a face acest lucru.

     Observație: în acest caz, nu este nevoie să folosim liste de inițializare. Dar este bine să vă familiarizați cu sintaxa lor, deoarece este singura posibilitate pentru a inițializa variabile membru care sunt `const`, care sunt referințe sau care nu au un constructor fără parametru (e.g. dacă am fi avut în structura noastră o variabilă membru de tip `ifstream`).

     Referințe utile: ce sunt și cum se folosesc listele de inițializare în C++.

   - O metodă statică `from_polar` care primește ca parametrii modulul și argumentul (unghiul) unui număr complex, construiește și returnează un nou obiect de tip `Complex` cu partea reală și partea imaginară corespunzătoare.

     Antetul acesteia ar trebui să fie:

     `static Complex from_polar(double modulus, double angle)`

     Iar ulterior o puteți apela în programul principal folosind sintaxa:

```
Complex z = Complex::from_polar(2, 0.5);
```

Reamintesc că un număr complex z se poate scrie în mod echivalent ca

$$z = x + iy = r(\cos\theta + i\sin\theta)$$

unde x este partea reală a numărului complex, y este partea lui imaginară, r este modulul numărului complex și θ este argumentul acestuia.

Observație: această metodă este un exemplu al pattern-ului static factory method. Deja avem un constructor pentru clasă cu signatura

```
Complex(double, double)
```

așa că nu am fi putut defini încă unul care să primească doi parametri de tip `double`. *Static factories* ne permit să avem niște constructori „cu nume”.

Referințe utile: variabile membru și metode statice, `<cmath>`.

- Supraîncărcați operatorul << pentru a permite afișarea unui număr complex, respectiv >> pentru a permite citirea de la tastatură a unui număr complex.

  Puteți găsi un exemplu de cum să faceți acest lucru aici. Antetul acestor funcții ar trebui să fie:

```
1 ostream& operator<<(ostream& out, const Complex& c);
2 istream& operator>>(istream& in, Complex& c);
```

  În cazul operatorului de afișare, este bine să primim obiectul de tip `Complex` printr-o referință constantă ca să evităm copierea lui și în același timp să nu-l modificăm din greșeală. În cazul operatorului de citire, trebuie să primiți obiectul prin referință ca să puteți să puteți actualiza datele din el.

  Operatorii ar putea fi definiți să returneze `void` în loc de `ostream&` (respectiv `istream&`), dar atunci nu am mai putea înlănțui operațiile de citire/afișare (e.g. nu am putea scrie `cin >> a >> b >> c`).

  După ce ați rezolvat acest subpunct, în programul principal ar trebui să puteți folosi operatorii ca în următorul exemplu:

```
1 Complex z;
2 cin >> z;
3 cout << z << '\n';
```

Observație: operatorii de citire/afișare pot fi supraîncărcați doar ca funcții în afara structurii/clasei; nu pot fi metode, deoarece vrem ca primul lor parametru să fie de tipul `istream&`/`ostream&`, nu obiectul implicit de tip `Complex` pe care îl primesc toate metodele.

Referințe utile: supraîncărcarea operatorilor <</>> în C++ (nu o să aveți nevoie de modificatorul `friend`, deoarece în acest caz datele membru sunt publice).

- Supraîncărcați operatorii `+`, `-`, `*`, respectiv `/`, ca să puteți aduna, scădea, înmulți și împărți numere complexe.

  Aceste funcții nu ar trebui să modifice numerele complexe pe care le primesc ca parametri, ci ar trebui să returneze niște obiecte noi.

  Observație: acești operatori îi puteți defini fie ca metode, caz în care vor primi un singur parametru de tip `Complex` (deoarece au acces deja la obiectul implicit), sau ca funcții libere, caz în care trebuie să primească doi parametrii de tip `Complex`.

  Referințe utile: supraîncărcarea operatorilor în C++.

- Supraîncărcați operatorii `==` și `!=` ca să puteți compara dacă două numere complexe sunt egale sau nu.

  Aceste funcții ar trebui să aibă un parametru de tip `Complex` (dacă sunt definite ca metode) sau doi parametri de tip `Complex` (dacă sunt definite ca funcții libere) și să returneze o valoare de tip `bool` (`true` sau `false`).

  Observație: dacă vreți să afișați rezultatul unei comparații, va trebui să scrieți `cout << (z1 == z2);` este obligatoriu să puneți paranteze, din cauza ordinii de evaluare a operatorilor în C++.

  Referințe utile: operatori de comparație, precedența operatorilor.

Scrieți subprogramul `main` corespunzător care să apeleze/testeze fiecare dintre metodele implementate.

8. Modificați structura definită la exercițiul anterior astfel încât să respecte *principiul encapsulării* (variabilele membru să fie private).

   Lăsați toate celelalte metode să fie publice. S-ar putea să fie nevoie să declarați cu modificatorul `friend` operatorii pe care i-ați supraîncărcat, ca să nu aveți erori de compilare.

Adăugați metode de tip getter și setter pentru partea reală, respectiv partea imaginară a numărului complex.

Referințe utile: modificatori de acces, funcții/clase „prieten” (modificatorul `friend`), exemplu de getter/setter pentru o variabilă membru privată.

9. Considerăm structura `Achizitie`, definită în felul următor:

```
struct Achizitie {
    int id;
    double valoare;
};
```

Definiți o altă structură `ReferintaAchizitie`, care să aibă ca date membru:

- un număr întreg constant, `const int id`;
- o referință la o achiziție, `Achizitie& achizitie`;

Puteți crea un obiect din această structură folosind constructorul fără parametrii?

Definiți un constructor pentru această structură care să primească ca parametru un obiect de tip `Achizitie&`. Va trebui să folosiți o listă de inițializare ca să inițializați datele membru.

Referințe utile: liste de inițializare în C++, inițializarea datelor membru constante, inițializarea datelor membru de tip referință.

10. În limbajul C, pentru a scrie un număr într-un fișier ar trebui să folosim următorul cod:

```
int numar = 1234;
FILE* f = fopen("fisier.txt", "w");
fprintf(f, "%d", numar);
fclose(f);
```

Observați că la final trebuie să apelăm `fclose` pentru a ne asigura că fișierul este salvat și închis cum trebuie. Dacă uităm să facem acest lucru, pot apărea bug-uri greu de diagnosticat.

Veți crea o clasă `MyFile` care să gestioneze o variabilă de tipul `FILE*`. Începeți prin a importa antetul `<cstdio>`, în care se află funcțiile menționate mai sus. Implementați:

- Un constructor care primește ca parametru un șir de caractere `const char nume_fisier[]`, reprezentând numele fișierului care va fi deschis. Acest constructor va folosi funcția `fopen` pentru a deschide fișierul indicat în modul de scriere și va salva pointerul returnat.

  Referințe utile: constructor cu parametri, `fopen`.

- Un destructor care apelează automat `fclose`.

  Referințe utile: destructor, `fclose`.

- O metodă `write` care primește ca parametru un număr întreg `numar` și îl scrie în fișier, lăsând un rând nou după, prin apelul `fprintf(f, "%d\n", numar);` (unde `f` este numele variabilei de tip `FILE*` din clasa voastră).

  Referințe utile: `fprintf`.

Creați o instanță a clasei `MyFile` în subprogramul principal și încercați să o folosiți ca să scrieți câteva valori într-un fișier.

Acest exercițiu este un exemplu foarte bun pentru utilitatea practică a constructorilor/destructorilor; vedeți și principiul *Resource Acquisition Is Initialization*.

11. Încercați să creați o copie a unei variabile de tip `MyFile` (clasa implementată la exercițiul anterior), de exemplu: `MyFile f("fisier.txt"); MyFile g(f);`

    Ce se întâmplă când se termină blocul în care sunt definite variabilele? Primiți o eroare?

    Constructorul de copiere generat implicit de compilator va copia variabila de tip `FILE*` în noua instanță a clasei. Când se rulează destructorii pentru ambele obiecte de tip `MyFile`, se va apela de două ori `fclose` pentru același fișier.

    Există moduri prin care putem elimina acest bug:

    - Definim constructorul de copiere, dar îl facem privat:

```
1 private:
2     MyFile(const MyFile&) {
3         // nu facem nimic aici
4     }
```

Asta va preveni copierea unui obiect de tip `MyFile` în afara clasei, dar în interiorul clasei încă există riscul să facem neintenționat acest lucru.

- (C++11) Folosim sintaxa „`= delete`” pentru a-i indica compilatorului că nu vrem să se genereze automat un constructor de copiere.

```
1     MyFile(const MyFile&) = delete;
```

Implementați o soluție asemănătoare și pentru operatorul =, pentru a preveni atribuirile de tipul `f = g;` pentru variabile de tip `MyFile` (acestea ar duce la același bug). Puteți fie să-l faceți privat, fie să folosiți sintaxa

```
1     void operator=(const MyFile&) = delete;
```

Referințe utile: ștergerea constructorilor de copiere și a operatorului =.

12. Nu vrem să permitem crearea unei copii a unui obiect de tipul `MyFile`, dar în unele cazuri ar fi util să putem transfera resursa deținută de el (variabila de tip `FILE*`) într-un alt obiect (de exemplu, ca să putem gestiona un vector de fișiere).

    În C++11, acest lucru se poate face definind constructorul/operatorul de mutare:

```
1 MyFile(MyFile&& other);
2 void operator=(MyFile&& other);
```

Aici apare un nou tip de referință (*`rvalue` reference*), care indică în fiecare caz că parametrul `other` a fost „preluat” de compilator și „mutat” în aceste funcții.

În cazul clasei noastre, fiecare dintre aceste funcții ar trebui să ia pointerul de tip `FILE*` din obiectul primit ca parametru, lăsându-l `NULL`/`nullptr` în urmă. Operatorul = ar trebui în plus să apeleze `fclose` pe pointerul pe care îl gestiona deja, dacă e cazul.

Va trebui să actualizați destructorul și metoda de afișare din `MyFile` ca să nu facă nimic în cazul în care variabila membru de tip `FILE*` este nulă.

Pentru a utiliza funcțiile nou-definite, putem folosi `std::move` din biblioteca `<utility>`. Exemplu de utilizare:

```
1 MyFile file1("..."), file2("...");
2 MyFile file3(move(file1));
3 file1 = move(file2);
```

`file3` ar trebui să scrie acum în fișierul deschis inițial de `file1`, `file1` ar trebui să scrie în fișierul deschis inițial de `file2`, iar `file2` ar trebui să nu scrie în niciun fișier (ar trebui să aibă nul pointerul la `FILE`).

Referințe utile: move semantics and rvalue references, rvalue references.

13. Biblioteca standard C++ oferă clasa `std::string` pentru gestionarea facilă a șirurilor de caractere alocate dinamic. Acest tip de date include multe funcționalități.

    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`.

14. Biblioteca standard C++ oferă clase și pentru gestionarea facilă a *colecțiilor* (vectorilor) de obiecte. 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[2]. 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>`.

[2]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.

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

  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`.