Materialul e realizat în cea mai mare parte de Gabriel Majeri.

Toate resursele pentru laborator: `https://bit.ly/fmi-laborator-poo-143`.

Indicații pentru *coding* mai plăcut

1. Folosiți nume sugestive pentru variabilele din rezolvările voastre.
2. Împărțiți logica rezolvărilor voastre în subprograme (funcții).
3. Încercați să înțelegeți erorile compilatorului, copy-paste-ul pe Google trebuie făcut selectiv pentru a fi util.
4. Folosiți `https://en.cppreference.com/w/` oricând sunteți nesiguri de utilizarea limbajului.

Exerciții

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

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

2. Probabil ați testat manual implementarea de la exercițiul anterior. Pentru acest exercițiu, scrieți un test automat folosind `assert` din header-ul `cassert` care să dovedească ca implementarea voastră este corectă.

   Referințe utile: assert în C++.

3. 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 condiționată `if...else`, operatorul modulo (%).

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

4. Reimplementați exercițiul anterior (FizzBuzz) astfel încât numerele nu vor fi afișate pe ecran, ci vor fi stocate într-un `array`. Scrieți și un test automat ca în exercițiul 2 care să vă valideze implementarea într-un subprogram separat și apelați-l din subprogramul `main`.

   Referințe utile: array-uri.

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

   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 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`;

6. Pentru exercițiul anterior, implementați multiple teste automate care să valideze ca implementarea voastră funcționează pe toate cazurile (numere negative, pozitive, valoarea zero etc).

   Notă: De acum înainte, validați-vă implementările folosind teste automate cu `assert` (acolo unde este posibil și relevant). Testarea automată este mult mai rapidă și mai puțin predispusă erorilor decat cea manuală.

7. 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: referințe în C++.

8. 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:

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

   Observație: în limbajul C nu există referințe. 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.

9. Scrieți un program în C++ care să citească de la tastatură un număr natural n și apoi n numere întregi, pe care să le salveze într-un vector alocat dinamic. Îl puteți crea scriind `int* vector = new int[n];` după ce l-ați citit pe n.

   Sortați vectorul folosind funcția `std::sort` din biblioteca standard și apoi afișați rezultatul pe ecran. Va fi nevoie să includeți header-ul `algorithm`, i.e. `#include <algorithm>`.

   Funcția `sort` primește doi parametrii: un pointer către primul element din vector, respectiv un pointer către elementul care ar veni fix după ultimul element din vector (nu e o problemă că acesta nu există).

   Apelul ar trebui să fie: `sort(&vector[0], &vector[n]);`

   La final, eliberați memoria alocată pentru vector prin `delete[] vector;` (trebuie să folosiți `delete[]`, nu doar `delete` simplu, pentru că este vorba de un întreg vector de elemente).

Referințe utile: alocarea dinamică de memorie în C++, cum se utilizează `std::sort`.

10. Definiți o nouă structură `Complex`, 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 era 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`, 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.

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.

  După ce ați rezolvat acest subpunct, în programul principal ar trebui să puteți scrie:

```
1 Complex z;
2 cin >> z;
3 cout << z;
```

  Observație: operatorii de 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.

  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.

De asemenea, scrieți subprogramul `main` corespunzător care să verifice că funcționează toate metodele implementate.

11. 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.

12. Definiți o clasă `IntVector` care să rețină un vector de numere întregi, alocat dinamic. Clasa ar trebui să aibă două variabile membru:

    - un număr întreg `int size`, reprezentând lungimea actuală a vectorului;
    - un pointer `int* data`, care reține adresa unei zone de memorie alocată dinamic, în care sunt salvate elementele vectorului.

    Implementați următoarele funcționalități:

    - Un constructor fără parametrii care să inițializeze pointer-ul `data` cu valoarea `NULL` sau (dacă folosiți C++11) `nullptr`, și să seteze `size` să fie 0.
    - Un constructor care permite crearea unui vector care constă din numărul întreg `x`, de `k` ori. Constructorul va avea ca parametrii `int k` și `int x`, în această ordine.

      Al doilea parametru ar trebui să aibă valoarea implicită 0.

      Constructorul va aloca dinamic memoria necesară pentru stocarea a `k` întregi, toți setați să aibă valoarea `x`.

      Referințe utile: setarea valorilor implicite pentru parametrii în C++.
    - Un destructor care dezalocă zona de memorie indicată de pointerul `data`, dacă acesta nu este `NULL`/`nullptr`.

      Referințe utile: destructor.
    - Un constructor de copiere care primește o referință constantă la un alt obiect din clasa `IntVector`, își alocă o nouă zonă de memorie de aceeași lungime cu cea a vectorului primit și copiază în ea valorile reținute de vectorul primit.

      Referințe utile: constructorul de copiere în C++, Rule of Three.
    - Supraîncărcați operatorul = în așa fel încât să permită asignarea prin copiere a unui vector în alt vector. Acest operator ar trebui să copieze datele din vectorul primit ca parametru (la fel ca și constructorul de copiere), doar că mai întâi ar trebui să dezaloce memoria deținută de vectorul destinație (dacă este cazul, dacă acesta nu este vid).

      Referințe utile: supraîncărcarea operatorului de atribuire/copiere.

- Supraîncărcați operatorul << ca să permiteți afișarea unui vector de numere întregi pe ecran (se vor afișa elementele acestuia, toate pe același rând, separate printr-un spațiu).
- Supraîncărcați operatorul >> ca să permiteți citirea unui vector de numere întregi dintr-un `istream`.

  Operatorul va citi mai întâi n, numărul de elemente care urmează să fie citite, iar apoi cele n numere întregi.

  Dacă vectorul primit ca parametru nu este gol, va trebui să eliberați mai întâi memoria reținută de el, iar apoi să-i alocați dinamic un array de dimensiune n în care să rețineți valorile citite.
- Definiți o metodă care să permită adăugarea unui nou element la sfârșitul vectorului.

  Va trebui să alocați dinamic o nouă zonă de memorie de dimensiune suficientă, să copiați vechile elemente în ea, să adăugați noul element la finalul ei și apoi să eliberați memoria pentru vechiul tablou alocat dinamic (dacă este cazul).

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

```c
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 cu parametrul `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`.

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

14. 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 `operator=`, pentru a preveni atribuirile de tipul `f = g;` pentru variabile de tip `MyFile` (acestea ar duce la același bug).

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

15. Implementați o clasă `MyString`, care să rețină un șir de caractere alocat dinamic (i.e. un pointer la un tablou unidimensional de `char`, alocat dinamic).

    Referințe utile: tipuri de date fundamentale, tipul de date pointer, cum se declară o clasă, specificatori de acces;

    Implementați următoarele funcționalități:

    - Crearea unui șir de caractere vid (acest lucru va fi indicat prin setarea pointer-ului intern la `NULL` / `nullptr`, ca să nu alocăm degeaba un vector de caractere de lungime 0);

      Referințe utile: constructori, pointerul `NULL`, `nullptr`.

    - Crearea unui șir de caractere pornind de la un string literal (să putem scrie `String sir("Un sir de caractere");`);

      Referințe utile: constructori cu parametri, string literals, operatorul `new[]`, `strlen`, `strncpy`;

    - Eliberarea automată a memoriei alocate;

      Referințe utile: operatorul `delete[]`;

    - Copierea unui șir în alt șir; acest lucru ar trebui să se poată face atât prin constructorul de copiere, cât și prin operatorul =.

      Referințe utile: constructorul de copiere, operatorul de atribuire prin copiere, constructorul de copiere versus operatorul de atribuire;

    (*) „Mutarea” unui șir în alt șir, noul șir preluând memoria gestionată de cel vechi, lăsând vid șirul sursă;

      Observație: acesta este un subpunct extra, veți avea nevoie de un compilator care suportă C++11 pentru a-l rezolva. Constructorul de mutare cel mai probabil nu va fi menționat la curs și nu intră la examen, dar este des folosit în practică.

      Referințe utile: move constructor;

    - Adăugarea unui caracter la șir;

    - Concatenarea a două șiruri;

    - Găsirea unui subșir într-un șir dat.

      Referințe utile: `strstr`.

    Scrieți și codul aferent în `int main()` care să apeleze și să testeze funcționalitățile de mai sus.

16. Implementați o clasă `IntList`, care să rețină o listă simplu înlănțuită de numere întregi (de tip `int`).

    Această clasă va avea definită în interiorul ei o clasă privată numită `Node`, care reprezintă un nod din listă.

    Referințe utile: nested class;

    Implementați următoarele funcționalități:

    - Crearea unei liste vide;
    - Crearea unei liste pornind de la un vector de numere întregi existent (i.e. să putem scrie `IntList lista({ 1, 2, 3, 4 });`);
    - Eliberarea automată a memorie alocate;
    - Copierea unei liste în altă listă;

    (*) Mutarea unei liste în altă listă, lăsând vidă lista inițială;

    - Adăugarea unui nou număr la finalul listei;
    - Adăugarea unui nou număr la începutul listei;
    - Concatenarea a două liste;
    - Accesarea elementului de pe poziția *i*;
    - Găsirea în listă a unui număr dat (dacă este sau nu prezent).