Exerciții cu alocarea manuală de memorie

1. 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`.

2. 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 (cât de mare este zona de memorie indicată de `data`);
   - un pointer `int* data`, care reține vectorul de numere întregi, alocat dinamic (reține adresa returnată de `new []`).

   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` numere întregi și va seta toate elementele să aibă valoarea `x`.

Referințe utile: alocarea dinamică de memorie în C++, 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 din vectorul primit.

  Antetul acestuia ar trebui să fie:

  ```
  IntVector(const IntVector& v)
  ```

  Observație: este obligatoriu să primim vectorul vechi prin referință (constantă). Dacă l-am primi prin valoare, la apel compilatorul ar folosi implicit constructorul de copiere, care apoi s-ar apela recursiv la infinit pe el însuși.

  Referințe utile: constructorul de copiere în C++.

- 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).

  Antetul operatorului = ar trebui să fie:

  ```
  IntVector& operator=(const IntVector& v)
  ```

  Primim prin referință constantă parametrul ca să evităm crearea unei copii în plus a acestuia și pentru a nu-l modifica din greșeală. Mai mult, la final ar trebui să returnăm o referință la obiectul implicit (i.e. `return *this`), ca să permitem utilizarea sintaxei de atribuire multiplă (e.g. `v1 = v2 = v3`).

  Referințe utile: supraîncărcarea operatorului de atribuire/copiere.

(*) Definiți un constructor de mutare și supraîncărcați operatorul = de mutare ca să permiteți mutarea memoriei deținute de un vector într-un alt vector. Vectorul destinație ar trebui să primească lungimea și pointerul

de la vectorul sursă, care va rămâne gol (va avea `size == 0` și `data == nullptr`).

- 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 (modificați vectorul actual, nu creați unul nou).

  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).

3. Alocați dinamic un vector de obiecte de tip `IntVector` (i.e. un vector de vectori). Puteți face acest lucru folosind sintaxa

   ```
   IntVector* vv = new IntVector[10];
   ```

   Încercați să vedeți cum/dacă au fost inițializați acești vectori (inspectați-i în debugger, sau afișați lungimile lor). Ce constructor a fost folosit?

   De asemenea, observați că atunci când apelați `delete[]` pe acest vector de obiecte, compilatorul apelează automat destructorul pentru fiecare obiect (puteți confirma acest lucru punând un *breakpoint* în destructor și rulând codul cu un debugger, sau afișând un mesaj în destructor).

   Referințe utile: cum inițializează `new[]` obiectele alocate, `delete[]` apelează destructorul pentru fiecare obiect.

4. Ce se întâmplă dacă încercăm să folosim operatorul `=` de copiere, definit pentru clasa `IntVector`, în cazul în care obiectul destinație și obiectul sursă sunt același (de exemplu, dacă am scrie `v = v`)?

   Tratați și această posibilitate în implementarea voastră. Verificați dacă pointerul `this` este egal cu pointerul `&v`; în acest caz, nu este nevoie să alocați

memorie sau să modificați datele membru.

Referințe utile: tratarea cazului de *self assignment*.

5. Supraîncărcați operatorul de indexare `[]` pentru una dintre clasele pe care le-ați implementat în exercițiile 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++.

6. Implementați o clasă `IntList`, care să rețină o listă simplu înlănțuită de numere întregi (de tip `int`), cu noduri alocate dinamic.

   Această clasă va avea definită în interiorul ei o clasă privată `Node`, care reprezintă un nod din listă. Fiecare nod va reține un număr întreg (valoarea acelui element din listă) și un pointer la următorul nod.

   Referințe utile: nested class;

   În clasa `IntList` este suficient să stocați un pointer la primul nod din listă pentru a o putea reține/accesa. Dacă doriți, puteți păstra și un pointer la ultimul nod din listă, pentru a adăuga mai ușor elemente la finalul ei.

   Implementați următoarele funcționalități:

   - Crearea unei liste vide (pointerii vor fi inițializați cu `NULL` / `nullptr`);

- Adăugarea unui nou număr la începutul listei (metoda ar trebui să funcționeze atât cu o listă inițial vidă, cât și cu una care are deja elemente în ea);
- Adăugarea unui nou număr la finalul listei (metoda ar trebui să funcționeze atât cu o listă inițial vidă, cât și cu una care are deja elemente în ea);
- Eliberarea automată a memorie alocate;

  Indicație: implementați destructorul pentru clasa `Node`, astfel încât să dea `delete` la următorul nod din listă (dacă acesta nu este nul). În clasa `IntList`, va fi suficient să dați `delete` doar la primul nod, iar destructorii se vor apela recursiv, ștergând toată lista.
- Crearea unei liste pornind de la un vector de numere întregi existent (i.e. să putem scrie `const int vector[] = { 1, 2, 3, 4 }; IntList lista(4, vector);`);
- Copierea unei liste în altă listă (folosind constructorul de copiere, respectiv supraîncărcând operatorul =);

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

- Concatenarea a două liste (cele două liste primite ca parametru trebuie să rămână neschimbate);
- Accesarea elementului de pe poziția *i* (se va returna valoarea acestuia);
- Setarea valorii elementului de pe poziția *i*;
- Găsirea în listă a unui număr dat (se va returna dacă este sau nu prezent).