Упражнение 5 — Конструктори, деструктори, динамични обекти

1. Реализирайте клас `Rational`, обектите на който описват рационално число. Обектите трябва да могат да бъдат конструирани чрез:

- конструктор, приемащ две цели числа — числител и знаменател;
- конструктор, приемащ едно цяло число, създаващ равно на него рационално;
- конструктор по подразбиране, създаващ числото 0.

Обектите трябва:

- да имат методи `add`, `subtract`, `multiply` и `divide`, всеки от които приема като аргумент рационално число и връща ново такова, без да променя обекта, върху който методът е бил извикан;
- метод `opposite`, който връща противоположното рационално число;
- методи `getNum` и `getDenom`, връщащи съответно числителя и знаменателя;
- метод `isPos`, връщащ дали числото е положително;
- метод `toDouble`, конвертиращ рационалното число към такова с плаваща запетая от тип `double`;
- метод `print`, принтиращ числото.

Изберете подходящо вътрешно представяне и подходящи сигнатура и връщан тип на методите, включително дали да са константни или не.

Допълнително условие*: Реализирайте класа така, че методите `getNum`, `getDenom` и `print` да отговарят на съкратения вид на числото. Изберете подходящо представяне за това.

2. Реализирайте клас RatDynamicArray, представящ масив от рационални числа с променлив брой елементи, реализиран чрез структурата динамичен масив. При тази структура:

- във всеки момент масивът има определен капацитет `capacity`, от който са заети толкова елемента, колкото е текущия брой елементи на масива — `size`;
- масивът е представен чрез динамично заделен масив от `capacity` на брой елемента, от които само първите `size` елемента съответстват на съществуващи рационални числа;
- при добавяне на елемент
    - ако има свободно място в динамично-заделения масив, то в зависимост от операцията, елемента се добавя в първата свободна клетка или на съответния индекс, като елементите от индекса до края се изместват предварително една клетка надясно;
    - иначе капацитетът се увеличава двойно, като се създава нов динамично-заделен масив с новия капацитет, а елементите на стария се

копират в новия, след което добавянето става аналогично на предната подточка;

Реализирайте подходящи:

- конструктор по подразбиране, създаващ празен масив с фиксиран начален капацитет (по ваш избор);
- копиращ конструктор;
- `operator=`
- деструктор
- метод `void push(const Rational& rational)`, добавящ рационално число в края на масива;
- метод `void insertAt(int index, const Rational& rational)`, добавящ рационално число на индекс `index`;
- метод `void pop()`, премахващ елемента на края на масива;
- метод `void removeAt(int index)`, премахващ елемента на индекс `index`;
- методи `Rational& get(int index)` и `const Rational& get(int index) const`, връщащи елемента, намиращ се на `index`;
- метод `void set(int index, const Rational& rational)`, променящ елемента на индекс `index`;
- метод `getSize()`, връщаш големината на масива.

При всички задачи приемете, че подаваните ви аргументи са коректни (например няма да се получи деление на нула, или индекса няма да излиза от размерите на масива).

Допълнителни трикове*

1. Стандартния начин за динамично заделяне на масив (например `new Rational[20]`) приема, че съответния клас притежава подразбиращ се конструктор. В нашия случай това е така, но можеше да поискаме да се създава рационално число само чрез числител и знаменател. Това може да се реши чрез така наречения оператор `placement new`, който създава обект във вече заделена памет (пропускайки заделянето на такава), което ни позволява да създадем обектите чрез, например, копиращия конструктор. Начинът на използване е следния:

```
// operator new[] заделя памет от определен брой байта,
// без да извиква конструктори
Rational* arr = (Rational*) operator new[](sizeof(Rational) *
capacity);

...

// добавяне на елемент rational:
new (arr + size++) Rational(rational);
```

```
...

// Освобождаване на паметта. Необходимо е ръчно да извикаме
// деструкторите, тъй като opreator delete[] не го прави:

for (int i = 0; i < size; i++) {
  arr[i].~Rational();
}

operator delete[](arr);
```

Опитайте да реализирате масива и по този начин.

2. Ако вече имаме написани копиращ конструктор и деструктор, то правилния начин да реализираме `operator=` е чрез тях (в противен случай ще имаме повтарящ се код и код, в който е доста вероятно да има грешка). Това става по следния начин:

- Пишем метод с swap, който прима неконстантен обект от същия клас и разменя всички полета на двата обекта. Примерна реализация за класа `RatDynamicArray` е:
  ```
  void swap(RatDynamicArray& arr) {
      std::swap(capacity, arr.capacity);
      std::swap(size, arr.size);
      std::swap(arr, arr.arr);
  }
  ```
- Тогава самия operator= приема следния вид:
  ```
  RatDynamicArray& operator=(const RatDynamicArray& other) {
      // проверка единствено за оптимизация, може и без нея
      if (this != &other) {
          swap(RatDynamicArray(other));
      }

      return *this;
  }
  ```
  Това работи по следния начин:
  - създава се временен обект, който е копие на подадения, използвайки копиращия конструктор (`RatDynamicArray(other)`)
  - стойността на този временен обект се разменя със стойността на нашия обект (през метода `swap` — той разменя всички член-данни);
  - накрая, след оценяване на израза, съдържащ извикването на swap, временния обект се унищожава, което води до извикване на неговия деструктор, а съответно и унищожаване на предишното състояние на нашия

обект (тъй като временния обект вече съдържа него).