Lecture review

A **class** in C++ can be used to implement an *abstract data type* (ADT). As discussed in lecture, the first step in designing an ADT is to decide upon the *API* i.e., the set of operations that can be performed on the data. The next step is to decide upon the *representation* of the data. The *instance variables* (also called *attributes*) of the class are used to represent the data. The representation of the data is hidden from the user of the ADT. The user can only access the data through the operations defined in the API. This is called *data encapsulation* or *information hiding*. The attributes and methods of a class are collectively called the *members* of the class.
A *constructor* is a special method that is used to initialize objects. It is called when an object of a class is created. It can be used to set initial values for object attributes. It can also be used to perform any other initialization tasks required when an object is created. A constructor is defined like a method, except that it has the same name as the class and has no return type. For example, the following constructors can be defined for a class implementing *rational* numbers.

- A *default constructor* that initializes the numerator to 0 and the denominator to 1.
- A constructor that takes a single integer argument and initializes the numerator to that value and the denominator to 1.
- A constructor that takes two integer arguments and initializes the numerator and denominator to those values.
- A *copy constructor* that takes a *rational* object as an argument and initializes the numerator and denominator to the values of the corresponding attributes of the argument object.

The following code shows the definitions of the four overloaded constructors described above.

```
class Rational {
    private:
    int num, den;

    public:
    // default constructor
    Rational() : num {0}, den {1} { /* empty */ }

    Rational(int n) : num {n}, den {1} { /* empty */ }

    Rational(int n, int d) : num {n}, den {d} { /* empty */ }

    // copy constructor
    Rational(const Rational& r) : num {r.num}, den {r.den} { /* empty */ }
};
```

Exercise 1

In this exercise, you will improve the class `Rational` defined in previous section by adding more methods. First, add the following constructors to the class `Rational`.

- A constructor that takes a `string` argument and initializes the numerator and denominator to the values specified in the string. For example, the string `"3/4"` should initialize the numerator to 3 and the denominator to 4.
- A constructor that takes a floating point number as an argument and initializes the numerator and denominator to the closest rational number to the argument. For example, the floating point number 0.75 should initialize the numerator to 3 and the denominator to 4.

Furthermore, add the following methods to the class `Rational`. Note that some of the following methods are declared as **`const`**, which means that the method does not modify the object on which it is called.

Method	Description
`int` `get_num()`**`const`**	Returns the numerator of the rational number.
`int` `get_den()`**`const`**	Returns the denominator of the rational number.
`void` `set_num(`**`int`** `n)`	Sets the numerator of the rational number to `n`.
`void` `set_den(`**`int`** `d)`	Sets the denominator of the rational number to `d`.
`string to_string()`**`const`**	Converts the rational number to `string` in the form `"num/den"`.
`void` `reduce()`	Reduces the rational number to its lowest terms. For example, if the rational number is 4/6, then it should be reduced to 2/3.
`Rational` **`operator+(const`** `Rational& r)`**`const`**	Returns the sum of the current rational number and the rational number `r` passed as an argument. The result should be in reduced form.
`Rational` **`operator-(const`** `Rational& r)`**`const`**	Returns the difference of the current rational number and the rational number `r` passed as an argument. The result should be in reduced form.
`Rational` **`operator*(const`** `Rational& r)`**`const`**	Returns the product of the current rational number and the rational number `r` passed as an argument. The result should be in reduced form.
`Rational` **`operator/(const`** `Rational& r)`**`const`**	Returns the quotient of the current rational number and the rational number `r` passed as an argument. The result should be in reduced form.
`bool operator==(const` `Rational& r)`**`const`**	Returns **`true`** if the current rational number is equal to the rational number `r` passed as an argument, and **`false`** otherwise. (`Rational(1, 2)== Rational(2, 4)` should return **`true`**)
`bool operator!=(const` `Rational& r)`**`const`**	Returns **`true`** if the current rational number is not equal to the rational number `r` passed as an argument, and **`false`** otherwise.

Write a client program that tests all the methods of the class `Rational`.

Exercise 2 .....

TicTacToe: Create a **class** `TicTacToe` that will enable you to write a complete program to play the game of tic-tac-toe. The class contains as private data a 3-by-3 two-dimensional array of integers. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place a 1 in the specified square. Place a 2 wherever the second player moves. Each move must be to an empty square.

The class should have the following public methods.

Method	Description
`TicTacToe()`	Default constructor. Initialize the board to all zeros.
void `clearBoard()`	Clear the board to all zeros.
void `to_string()const`	Return a string representation of the current board.
int `gameStatus()const`	Returns `1` or `2` if the player 1 or 2 wins respectively, `3` if the game is draw, and `0` if the game is not over yet.
bool `move(int player, int row, int col)`	Play a move for the player `player` at the row `row` and column `col`. Returns **true** if the move is successful, and **false** if the square is not empty.

If you feel ambitious, add a method **void** `autoMove(int player)` so that the computer makes the moves for `player`.

Write a client that runs a complete game of tic-tac-toe. Allow the two players to alternatively make the moves. After each move, determine whether the game has been won or is a draw. Following is a sample run of the program:

```
Welcome to Tic-Tac-Toe!

Play 1 move: enter row and column: 0 0
            1 | 0 | 0
           -----------
            0 | 0 | 0
           -----------
            0 | 0 | 0

Play 2 move: enter row and column: 1 1
            1 | 0 | 0
           -----------
            0 | 2 | 0
           -----------
            0 | 0 | 0
```

```
Play 1 move: enter row and column: 0 1
            1 | 1 | 0
           -----------
            0 | 2 | 0
           -----------
            0 | 0 | 0

Play 2 move: enter row and column: 1 2
            1 | 1 | 0
           -----------
            0 | 2 | 2
           -----------
            0 | 0 | 0

Play 1 move: enter row and column: 0 2
            1 | 1 | 1
           -----------
            0 | 2 | 2
           -----------
            0 | 0 | 0

Player 1 wins!
```

Exercise 3 .....

Create class `IntegerSet` for which each object can hold integers in the range 0 through 100. Represent the set internally as a `vector` of **`bool`** values. Element `a[i]` is **`true`** if integer `i` is in the set. Element `a[j]` is **`false`** if integer `j` is not in the set.

The default constructor initializes a set to the so-called "empty set," i.e., a set for which all elements contain **`false`**.

- Provide member functions for the common set operations. For example, provide a `unionOfSets` member function that creates a third set that is the set-theoretic union of two existing sets (i.e., an element of the result is set to **`true`** if that element is **`true`** in either or both of the existing sets, and an element of the result is set to **`false`** if that element is **`false`** in each of the existing sets).
- Provide an `intersectionOfSets` member function which creates a third set which is the set-theoretic intersection of two existing sets (i.e., an element of the result is set to **`false`** if that element is **`false`** in either or both of the existing sets, and an element of the result is set to **`true`** if that element is true in each of the existing sets).
- Provide an `insertElement` member function that places a new integer `k` into a set by setting `a[k]` to **`true`**. Provide a `deleteElement` member function that deletes integer `m` by setting `a[m]` to **`false`**.
- Provide a `to_string` member function that returns a set as a `string` containing a list of numbers separated by spaces. Include only those elements that are present in the set (i.e., their position in the vector has a value of **`true`**). Return empty string `""` for an empty set.
- Provide an `isEqualTo` member function that determines whether two sets are equal.
- Provide an additional constructor that receives an array of integers, and uses the array to initialize a set object.

Now write a driver program to test your `IntegerSet` class. Instantiate several `IntegerSet` objects. Test that all your member functions work properly.