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 and destructor 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.
- A **copy assignment operator** that properly handles assignment between existing objects by copying the values of the numerator and denominator. It should also check for self-assignment to prevent unnecessary operations.
- A **destructor** is a special member function that is automatically called when an object goes out of scope or is explicitly deleted, and its purpose is to clean up any resources allocated by the object (if necessary).

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 */ }

    // Copy Assignment Operator
    Rational& operator=(const Rational& r) {
    if (this == &r) return *this; // Handle self-assignment
    num = r.num;  // Directly copy numerator
    den = r.den;  // Directly copy denominator
    return *this;
}

    // destructor
    ~Rational() {}
    /*Since there is no dynamic memory allocation (i.e., num and den are basic
       types), the destructor is empty and doesn't need to perform any special
       cleanup.*/
};
```

Lab exercises

Exercise 1 .....

Rational Numbers: Modify the `Rational` class so that it dynamically allocates memory for the numerator and denominator instead of using simple integer variables (*num and den will now become pointers*). The class should represent rational numbers in their simplest form and provide arithmetic operations.

- Implement a **constructor** that dynamically allocates memory for the numerator and denominator and ensures the fraction is stored in its **simplest form**.
- Implement a **copy constructor** to create deep copies of objects.
- Implement a **copy assignment operator** to handle assignment between objects properly.
- Implement a **destructor** to release dynamically allocated memory.

Example Usage:

```
Rational r1(3, 4);  // Calls Parameterized Constructor
Rational r2 = r1;   // Calls Copy Constructor
Rational r3;        // Calls Default Constructor
r3 = r1;            // Calls Copy Assignment Operator
```

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 by dereferencing the dynamically allocated memory (`return *num;`).
`int get_den() const`	Returns the denominator of the rational number by dereferencing the dynamically allocated memory (`return *den;`).
`void set_num(int n)`	Sets the numerator by updating the value in dynamically allocated memory (`*num = n;`).
`void set_den(int d)`	Sets the denominator by updating the value in dynamically allocated memory (`*den = d;`).
`string to_string() const`	Converts the rational number to a string in the form `"num/den"` by dereferencing pointers (`return std::to_string(*num) + "/" + std::to_string(*den);`).
`void reduce()`	Reduces the rational number to its lowest terms by computing the greatest common divisor (GCD) and updating `*num` and `*den` accordingly.
`Rational operator+(const Rational& r) const`	Returns the sum of the current rational number and `r` by dynamically allocating a new Rational object with the computed sum.
`Rational operator-(const Rational& r) const`	Returns the difference of the current rational number and `r` by dynamically allocating a new Rational object with the computed difference.
`Rational operator*(const Rational& r) const`	Returns the product of the current rational number and `r` by dynamically allocating a new Rational object with the computed product.
`Rational operator/(const Rational& r) const`	Returns the quotient of the current rational number and `r` by dynamically allocating a new Rational object with the computed quotient.
`bool operator==(const Rational& r) const`	Returns `true` if the current rational number is equal to `r` by comparing `*num` and `*den` values.
`bool operator!=(const Rational& r) const`	Returns `true` if the current rational number is not equal to `r` by comparing `*num` and `*den` values.
`Rational(const Rational& r)`	Copy constructor that creates a deep copy of another rational number by allocating new memory and copying values.
`Rational& operator=(const Rational& r)`	Copy assignment operator that ensures deep copying by assigning new values to already allocated memory.
`~Rational()`	Destructor that frees the dynamically allocated memory for `num` and `den` using `delete`.

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

Exercise 2 .....

IntegerSet Create a class `IntegerSet` for which each object can hold integers in the range 0 through 100. Represent the set internally as a dynamically allocated boolean array (`bool*`). Element `a[i]` is `true` if integer `i` is in the set, and `false` otherwise.

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

- Implement a **constructor** that dynamically allocates memory for a boolean array of size 101 and initializes all values to `false`.
- Implement a **copy constructor** to ensure deep copying of dynamically allocated memory.
- Implement a **copy assignment operator** to handle assignment between objects properly.
- Implement a **destructor** to properly free the allocated memory.
- Provide member functions for the common set operations:
  - `unionOfSets(const IntegerSet&)`: Creates a third set that is the set-theoretic union of two existing sets.
  - `intersectionOfSets(const IntegerSet&)`: Creates a third set that is the set-theoretic intersection of two existing sets.
  - `insertElement(int k)`: Inserts a new integer `k` into a set by setting `a[k]` to `true`.
  - `deleteElement(int m)`: Deletes integer `m` by setting `a[m]` to `false`.
  - `to_string()` Returns a string containing a list of numbers separated by spaces. Include only those elements that are present in the set.
  - `isEqualTo(const IntegerSet&)`: Determines whether two sets are equal.
- Provide an additional constructor that receives an array of integers and its size, and uses this array to initialize a set object.

Example Usage:

```
IntegerSet set1;
set1.insertElement(10);
IntegerSet set2 = set1;  // Copy constructor is called
IntegerSet set3;        // Calls Default Constructor
set3 = set1;            // Calls Copy Assignment Operator
```

Write a driver program to test your `IntegerSet` class. Instantiate several `IntegerSet` objects and test all member functions to ensure correct functionality and proper dynamic memory management.