OOBJECT ORIENTED PROGRAMMING
Assignment: 4

Deadline: April 8, 2019 3:00 PM

Attention

- Make sure that you read and understand each and every instruction. If you have any questions or comments, you are encouraged to discuss with your instructors (and colleagues) on Piazza.
- Plagiarism is strongly forbidden and will be very strongly punished. If we find that you have copied from someone else or someone else has copied from you (with or without your knowledge) both of you will be punished. You will be awarded straight zero in this assignment or all the assignments.
- Submit two files "*.h, *.cpp" file for each question of your assignment.
- **Note: We will be running your code against our test cases, and a test case failure or a segmentation fault/incorrect result will result in loss of marks.**

Rational

A number that can be made by dividing two integers (an integer is a number with no fractional part) is called a rational number. The word comes from "ratio".

Your goal is to overload the operators for a generic "Rational" class. The Rational class will support mixed types in expressions e.g. 0.5 as "1/2", 0.25 as "1/4" etc. You will need to write two files (Rational.h and Rational.cpp). Your implemented class must fully provide the definitions of following class (interface) functions. You are required to implement the concepts related to operator overloading learned during the course.

```
class Rational {
// think about the private data members
public:
Rational(int n=0,int d=1);
Rational(const Rational &copy);// copy constructor to initialize the Rational form existing
Rational //object

// Binary Operators
// Assignment Operator
Rational operator = (const Rational &x);
// Arithmetic Operators
Rational operator+(const Rational &x) const;
Rational operator-(const Rational &x) const;
Rational operator*(const Rational &x) const;
Rational operator/(const Rational &x) const;
// Compound Arithmetic Operators
Rational operator += (const Rational &x);
Rational operator -= (const Rational &x);
Rational operator *= (const Rational &x);
Rational operator /= (const Rational &x);
// Logical Operators
bool operator == (const Rational & other) const;
```

```
bool operator < (const Rational & other) const;
bool operator > (const Rational & other) const;
bool operator <= (const Rational & other) const;
bool operator >= (const Rational & other) const;

// Unary Operator
// Conversion Operator
operator string() const; // returns 2/3 as “2/3”. If the denominator is 1 then only the
numerator is returned, i.e. for 2/1 the operator shall return “2”
~Rational(); // destructor
};

// Operator Overloading as Non-Member Functions
// Stream Insertion and Extraction Operators
ostream& operator<<(ostream& output, const Rational &);    // outputs the Rational
istream& operator>>(istream& input, Rational&);     // inputs the Rational
```

String

Your goal is to overload the operators for the “**String**” class that you have implemented in your first assignment. You will need to write two files (String.h and String.cpp). Your implemented class must fully provide the definitions of following class (interface) functions.

```
class String {
 // think about the private data members
public:
// provide definitions of following functions
String(); // default constructor
String(const char *str); // initializes the string with constant c-string
String(const String &); // copy constructor to initialize the string from the existing
string
String(int x); // initializes a string of predefined size

// Binary Operators
// Sub-script Operators
char &operator[](int i); // returns the character at index [x]
const char operator[](int i) const; // returns the character at index [x]
// Arithmetic Operators
String operator+(const String &str) const; // appends a String at the end of the String
String operator+(const char &str) const; // appends a char at the end of the String
String operator+(char *&str) const; // appends a String at the end of the String
String operator-(const String &substr) const; //removes the substr from the String
String operator-(const string &substr) const; //removes the substr from the String
// Assignment Operators
String& operator=(const String&); // copies one String to another
String& operator=(char*); // copies one c-string to another
String& operator=(const string&); // copies one string to another
// Logical Operators
bool operator==(const String&) const; // returns true if two Strings are equal
bool operator==(const string&) const; // returns true if the string is equal to the String
bool operator==(char *) const; // returns true if the c-string is equal to the String
```

```
// Unary Operators
// Boolean Not Operator
bool operator!(); // returns true if the String is empty
// Function-Call Operators
int operator()(char) const; // returns the index of the character being searched
int operator()(const String&) const; // returns the index of the String being searched
int operator()(const string&) const; // returns the index of the string being searched
int operator()(char *) const; // returns the index of the c-string being searched
// Conversion Operator
operator int() const; // returns the length of string
~String(); // destructor
};

ostream& operator<<(ostream& output, const String&); // outputs the string
istream& operator>>(istream& input, String&); // inputs the string
```

Big Integer

BigInt class is used for the mathematical operations that involve very big integer calculations that are outside the limit of all available primitive data types. For example, factorial of 100 contains 158 digits in it so we can't store it in any primitive data type available. We can store as large Integer as we want in it.

Your goal is to overload the operators for a generic "BigInt" class. You will need to write two files (BigInt.h and BigInt.cpp). Your implemented class must fully provide the definitions of following class (interface) functions .

```
class BigInt
{
//think about the private data members
public:
BigInt(int val = 0);
BigInt(const string& text);
BigInt(const BigInt& copy); // copy constructor

// Binary Operators
// Arithmetic Operators
BigInt operator+(const BigInt& val) const;
BigInt operator+(int val) const;
BigInt operator-(const BigInt& val) const;
BigInt operator-(int val) const;
BigInt operator*(const BigInt& val) const;
// Compound Assignment Operators
BigInt operator+=(const BigInt& rhs);
BigInt operator-=(const BigInt& rhs);
BigInt operator*=(const BigInt& rhs);
// Logical Operators
bool operator==(const BigInt& val) const;
bool operator!=(const BigInt& val) const;
bool operator<(const BigInt& val) const;
bool operator<=(const BigInt& val) const;
bool operator>(const BigInt& val) const;
bool operator>=(const BigInt& val) const;
```

```
// Unary Operators
BigInt& operator++();  // Pre-increment Operator
BigInt operator++(int);  // Post-increment Operator
BigInt& operator--();  // Pre-decrement Operator
BigInt operator--( int ); // Post-decrement Operator

//Conversion Operator
operator string();    // return value of the BigInt as string
~BigInt(); // destructor
};

ostream& operator<<(ostream& output, const BigInt& val); // outputs the BigInt
istream& operator>>(istream& input, BigInt& val); // inputs the BigInt
```

Polynomial

Your goal is to overload the operators for a generic "**Polynomial**" class. A polynomial will be represented via its coefficients. Here is a third degree polynomial $4x^3 + 3x + 2$; where we have four coefficients and the coefficient corresponding to 2nd power is zero. You are required to write two files (Polynomial.h and Polynomial.cpp). Your implemented class must fully provide the definitions of following class (interface) functions.

```
class Polynomial {
// think about the private data members (coefficient values can be of type int)
public:
//include all the necessary checks before performing the operations in the functions
Polynomial(); // a default constructor
Polynomial(int); // a parameterized constructor, received the highest degree of polynomial
Polynomial(const Polynomial &); // a copy constructor

// Binary Operators
// Assignment Operator
Polynomial operator=(const Polynomial& rhs); //assigns (copies) the rhs Polynomial to "this"
Polynomial
// Arithmetic Operators
Polynomial operator+(const Polynomial &); // adds two Polynomials and returns the result
Polynomial operator-(const Polynomial &); // subtracts two Polynomials and returns the
result
// Compound Assignment Operators
void operator+=(const Polynomial&); // adds two Polynomials
void operator-=(const Polynomial&); // subtracts two Polynomials
// Logical Operator
bool operator==(const Polynomial &); // compares and returns true if equal

// Conversion Operator
operator string() const; // returns the value of the Polynomial as a string like "4x^3 + 3x
+ 2"
~Polynomial(); // destructor
};
```

```
ostream& operator<<(ostream& output, const Polynomial&); // outputs the Polynomial
istream& operator>>(istream& input, Polynomial&); // inputs the Polynomial
```

Bouquet of **Flowers**

Your goal here is to write classes for creating a bouquet of flowers. To create the bouquet of flower you will need to write following two classes.

Design a class Flower. A "Flower" is characterized by the following attributes:

- a name
- a color
- a basic price per unit
- an indication whether the flower is perfumed or not
- and an indication to know whether the flower is on sale.

The class has the following behaviors:

- a constructor initializing the attributes using parameters given in the order shown by the provided main(); a default constructor will not be necessary but the last two parameters will have false as default value
- a **price()** method returning the flower's price: the price will be the base price if the flower is not on sale; otherwise, the price will be half the base price
- a bool **perfume()** method indicating whether the flower is perfumed or not
- **operator string() const** to return value of the Flower as a string as:
  <Name> <Color> <Perfumed>, Price: <Price> Rs.
- Overloaded stream insertion operator. The characteristics have to be displayed in strict accordance with the following format:
  <Name> <Color> <Perfumed>, Price: <Price> Rs.
- an overloading of the == operator returning true if two flowers are identical, false otherwise. Two flowers are considered identical if they have the same name, color, and the two flowers are both either perfumed or not (neither the price nor the fact that the flower is on sale or not is involved in the comparison).

Next, write a "Bouquet" class which will be modeled using a dynamic array of Flowers.

The Bouquet class offers the following methods:

- a method bool **perfume()** returning true if the bouquet is perfumed and false otherwise; a bouquet is perfumed if at least one of its flowers is perfumed;
- a method **price()** without parameters returning the price of the bouquet of flowers; This is the sum of the prices of all its flowers; this sum is multiplied by two if the bouquet is perfumed;
- **operator string() const** to return value of the Bouquet as a string as:
  If the bouquet does not contain any flower,

Still no flower in the bouquet
Else:
<Flower1>
..
<FlowerN>
Total Price: <Price_of_bouquet> Rs.

- a stream insertion method, should display all information of bouquet with the total price. This method will display the characteristics of the bouquet of flowers respecting rigorously the following format:
  If the bouquet does not contain any flower,
  Still no flower in the bouquet
  Else:
  Perfumed Bouquet composed of:

  <Flower1>
  ..
  <FlowerN>
  Total Price: <Price_of_bouquet> Rs.

Here < FlowerX > means display of the X^{th} flower of the bouquet in the format specified by the overload of the << operator. There is a newline after displaying each flower and after displaying the price of the bouquet.

- an overload of the += operator which allows adding a flower to the bouquet, the flower will always be added at the end.
- an overload of the -= operator taking as a parameter a flower and removing from the bouquet all the flowers identical to the latter (according to the definition of the == operator);
- an overloaded + operator according its usage in the provided main
- an overloaded - operator according to its usage in the provided main

---

```
int main() {
	// example of Yellow oderless rose.
	Flower r1("Rose", "Yellow", 1.5);
	cout << r1 << endl;
	// example of Yellow perfumed rose
	Flower r2("Rose", "Yellow", 3.0, true);
	// example of perfumed Red rose on sale
	Flower r3("Rose", "Red", 2.0, true, true);
	Bouquet b1;
	b1 += r1; // add one Flower of r1 type
	b1 += r1; // add another Flower of r1
	b1 += r2;
	b1 += r3;
```

```
        cout << b1 << endl;

        b1 = b1 - r1; // Delete all the Flowers of type r1
        cout << b1 << endl;

        Bouquet b2;
        b2 = b1 + r1; // Add one Flower of type r1
        cout << b2 << endl;

        // Delete all the perfumed flowers from the bouquet.
        b2 -= r2;
        b2 -= r3;
        cout << b2;
        return 0;
}
```
