Deadline:

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 GCR.
- 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.**

Dollar

The US dollar is composed of many different coins, which include nickels, cents, and quarters, which can be combined to make up any amount of money. In this question, you are required to create a Money Class in which a specific amount of money will be represented in terms of dollars, quarters, nickels, and cents in that precedence. This money class will consist further of the dollar, quarter, nickel, and cent classes.

For example, if your amount of money is 4.56, your class will first store the maximum possible amount in dollars, then it will move on to quarters, nickels, and cents. You are required to implement the following functions:

```
class Money {
// think about the private data members
public:
Money();// default constructor
Money(double);// parameterized constructor

//Implement getters and setter functions

Money operator+(Money m); //
Money operator-(Money m); //
Money operator+(Dollar d): //
Money operator-(Dollar d); //
Money operator+(Nickel d): //
Money operator-(Nickel d); //
Money operator+(Quarter d): //
Money operator-(Quarter d); //
Money operator+(Cent d): //
Money operator-(Cent d); //

//These are all addition and subtraction on overloads to add specific coins to the total
amount of money


Money operator ++() //Round up the current amount of money to the nearest quarter
Money operator --() //Round down the current amount of money to the nearest quarter
```

```
Bool operator>=()
Bool operator<=()
//Comparison operators
Quarters operator!()

//Returns the maximum number of quarters that can be obtained from Money
Nickels operator~()

//Returns the maximum number of nickels that can be obtained from Money

Money operator/(int n)

//Returns the money object created if the current money amount were divided into n
parts

Money operator*(int n)
//Returns the money object created by multiplying the current total amount by n

};
```

Date

Write a class called Date that represents a date consisting of a year, month, and day(you can think about private data members of the class). A Date object should have the methods and operators which are shown in Table 1:

Table 1

Date(int year. int month, int day)	Constructs a new Date object to represent the given date.
operator =	Overload = operator to assign values
d2=d1+1	Overload + operator which takes integer as argument It moves this Date object forward in time by the given number of days.
d2=di-4	Overload - operator which takes integer as argument It moves this Date object backward in time by the given number of days.
d3=d1+d2	Overload + operator which takes Date object as argument.
d3=d1-d2	Overload - operator which takes Date object as argument.
bool a=d1>d2:	Overload > operator which returns true or false.
bool a=d1>=d2:	Overload >= operator which returns true or false.
bool a=d1	Overload < operator which returns true or false.
bool a=dl<=d2:	Overload <= operator which returns true or false.
bool d1!=d2:	Overload != operator which returns true or false.
bool d1==d2:	Overload == operator which returns true or false.
int getDay()	Returns the day value of this date: for example, for the date 2006/07/22 returns 22.
int getMonth()	Returns the month value of this date; for example, for the date 2006/07/22 returns 7.
int getYear()	Returns the year value of this date: for example, for the date 2006/07/22 returns 2006.
bool isLeapYear()	Returns true if the year of this date is a leap year. A leap year occurs every four years, except for multiples of 100 that are not multiples of 400. For example, 1956. 1844, 1600, and 2000 are leap years, but 1983. 2002. 1700. and 1900 are not.
String toString()	Returns a String representation of this date in year/month/day order. such as "2006/07/22"

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;
}
```
