CMPT 135: Lab Work Week 5-6

1. Is it possible to have a different name for the destructor member function instead of the standard C++ naming **~CLASS_NAME**? That is does C++ language syntax allow a different name? If yes, what is the shortcoming of such a change?

2. Consider the following program. Draw a memory diagram to show both the main stack and the heap memory configurations as the program code is executed.

```
int main()
{
	SmartArray *x;
	x = new SmartArray();
	for (int i = 0; i < 5; i++)
		x->append(i+6);
	cout << "*x is " << *x << endl;

	//Delete any memory that was allocated on the heap memory

	system("Pause");
	return 0;
}
```

   Assuming that the class has a destructor member function, our aim is now to delete any memory that had been allocated on the heap memory. How can you delete the memory?

   Answer: The simplest way would be to do it in two steps as follows:

```
x->~SmartArray();	//Delete the heap memory containing the elements
delete x;		//Delete the memory pointed to by x
```

   However, actually we don't need the first line of code. **`delete x;`** is enough. Why? Because it will automatically call the destructor first and then delete the memory pointed to by x.

3. Consider the **SmartArray** class we have developed. Assume we have the destructor implemented as we have seen in lecture; but we do not have the copy constructor overloaded instead we will use the default C++ copy constructor. Now, the following main program will work fine but it will crash after the main program **finishes** execution. Why?

```
int main()
{
	SmartArray L1;
	for (int i = 0; i < 5; i++)
		L1.append(i+6);
	SmartArray L2 = L1;
	cout << "L1 is " << L1 << endl;
	cout << "L2 is " << L2 << endl;
	system("Pause");
	return 0;
}
```

4. Consider the **SmartArray** class we have developed. Assume we have the destructor implemented as we have seen in lecture; but we do not have the assignment operator overloaded instead we will use the default C++ assignment operator. Now, the following main program will work fine but it will crash after the main program **finishes** execution and also it will have **memory leak**. Why?

```
int main()
{
	SmartArray L1, L2;
	for (int i = 0; i < 5; i++)
	{
		L1.append(i+6);
		L2.append(10-i);
	}
	cout << "L1 is " << L1 << endl;
	cout << "L2 is " << L2 << endl;
	L2 = L1;
	cout << "L1 is " << L1 << endl;
	cout << "L2 is " << L2 << endl;
	system("Pause");
	return 0;
}
```

5. Analyze the following class and testing main function carefully and give the output that would be printed when you run the program. Please do not type the code. Instead, manually trace the program on paper and determine the output. Afterwards, type the code and run it. See if you get same output on your paper. If not analyze it again making sure you step through each statement carefully.

```
class Fruit
{
private:
    int weightInGrams;
    string color;
    static int counter;
public:
    Fruit():weightInGrams(0), color("")
    {
	cout << "Inside default constructor." << endl;
	counter++;
    }
    Fruit(const int &w, const string &c):weightInGrams(w), color(c)
    {
	cout << "Inside non-default constructor." << endl;
	counter++;
    }
    Fruit(const Fruit &f):weightInGrams(f.getWeight()), color(f.getColor())
    {
	cout << "Inside copy constructor." << endl;
	counter++;
    }
    ~Fruit()
    {
	cout << "Inside destructor." << endl;
	counter--;
    }
    Fruit& operator = (const Fruit &f)
    {
	cout << "Inside assignment operator." << endl;
	setWeight(f.getWeight());
	color = f.getColor();
	return *this;
    }
```

```
    int getWeight() const
    {
            return weightInGrams;
    }
    string getColor() const
    {
            return color;
    }
    void setWeight(const int &w)
    {
            weightInGrams=w;
    }
    void setColor(const string &c)
    {
            color = c;
    }
    static int getCounter()
    {
            return Fruit::counter;
    }
};
int Fruit::counter = 0;
const Fruit& foo1(const Fruit f1, const Fruit &f2, const Fruit *f3)
{
    cout << "In foo1, starting with " << Fruit::getCounter() << " fruits." << endl;
    Fruit f = f1;
    if (f2.getWeight() > f.getWeight())
            f = f2;
    if (f3->getWeight() > f.getWeight())
            f = *f3;
    cout << "In foo1, finishing with " << Fruit::getCounter() << " fruits." << endl;
    return f2;
}
Fruit foo2(const Fruit f1, const Fruit &f2, const Fruit *f3)
{
    cout << "In foo2, starting with " << Fruit::getCounter() << " fruits." << endl;
    Fruit f = f1;
    if (f2.getWeight() > f.getWeight())
            f = f2;
    if (f3->getWeight() > f.getWeight())
            f = *f3;
    cout << "In foo2, finishing with " << Fruit::getCounter() << " fruits." << endl;
    return f;
}
int main()
{
    Fruit f1;
    f1.setWeight(5);
    f1.setColor("Green");
    Fruit f2(10, "Yellow");
    Fruit f3 = f2;
    cout << "In main before foo1, there are " << Fruit::getCounter() << " fruits." << endl;
    Fruit f4 = foo1(f1, f2, &f3);
    cout << "In main after foo1, there are " << Fruit::getCounter() << " fruits." << endl;
    cout << "In main before foo2, there are " << Fruit::getCounter() << " fruits." << endl;
    Fruit f5 = foo2(f1, f2, &f3);
    cout << "In main after foo2, there are " << Fruit::getCounter() << " fruits." << endl;
    system("Pause");
    return 0;
}
```

6. Consider the **SmartArray** class we implemented during the lecture. Add a member function named **findElement** that takes an integer argument and returns the index of the first element of the calling object

that is equal to the argument if the integer argument is found in the calling object and returns -1 otherwise.

7. Consider the **SmartArray** class we implemented during the lecture. Add a member function named **remove** that takes an integer argument and removes the first element of the calling object that is equal to the integer argument; if such an element is found in the calling object. Design this function to return true if an element is removed from the calling object and return false otherwise.

   For example if the calling object is **L = [1, 2, 3, 2, 5]** then

   L.remove(2) must modify L to **L = [1, 3, 2, 5]** and return true, and

   L.remove(4) must not modify L and return false.

8. Consider the **SmartArray** class we implemented during the lecture. Add a member function named **removeAll** that takes an integer argument and removes all the occurrences of the argument in the calling object. Hint:- Use your remove function.

9. Consider the **SmartArray** class we implemented during the lecture. Add a binary operator **+** member function that has a **SmartArray** operand on the left hand side and a **SmartArray** operand on the right hand side; and that returns a **SmartArray** made up of the concatenation of the **SmartArray**s.

   Example L1 = [1, 2, 3] and L2=[2, 5]. Then L3 = L1 + L2; should result to L3 = [1, 2, 3, 2, 5]

10. Consider the **SmartArray** class we implemented during the lecture. Add a binary operator **==** member function that has a **SmartArray** operand on the left hand side and a **SmartArray** operand on the right hand side; and that returns true if both the operands have the same size and identical corresponding elements and returns false otherwise.

    Example L1 = [1, 2, 3] and L2=[1, 2, 3]. Then L1 == L2; should return true

    Example L1 = [1, 2, 3] and L2=[1, 4, 3]. Then L1 == L2; should return false

11. Consider the **SmartArray** class we implemented during the lecture. Add a binary operator **!= member function** that has a **SmartArray** operand on the left hand side and a **SmartArray** operand on the right hand side; and that returns the negation of the == operator described above.

    Example L1 = [1, 2, 3] and L2=[1, 2, 3]. Then L1 != L2; should return false

    Example L1 = [1, 2, 3] and L2=[1, 3]. Then L1 != L2; should return true

12. Add the unary operators **++** and **- -** to the **SmartArray** class that will have the following functionalities:

    - Pre-increment operator that increments all the elements in the dynamic array by 1
    - Pre-decrement operator that decrements all the elements in the dynamic array by 1
    - Post-increment operator that increments all the elements in the dynamic array by 1
    - Post-decrement operator that decrements all the elements in the dynamic array by 1

13. Add the unary operator **–** to the **SmartArray** class. This operator must not modify the calling object; instead it returns a new **SmartArray** object whose elements are the elements of the calling object multiplied by -1.

14. Add a binary operator **–** member function to the **SmartArray** class. This operator has **SmartArray** object as its left hand side operand and a **SmartArray** object as its right hand side operand (for example **A – B**) and it

must return a SmartArray object containing the **distinct** elements that are in **A** but not in **B** together with those in **B** but not in **A** . For example if **A = [5, 1, 3, 2, 1, 4, 2]** and **B = [3, 9, 5, 8, 9]** then **A – B** must return **[1, 2, 4, 9, 8]**.

15. Write a C++ program that generates a random integer **n** in the range [10, 25], constructs an empty SmartArray, and then appends **n** random integers in the range [-15, 15] to the SmartArray object. Finally print the minimum and maximum elements of the SmartArray object.

16. Write a C++ program that generates a random integer **n** in the range [10, 25], constructs an empty SmartArray, and then appends **n** user input integers to the SmartArray object. Finally print the minimum and maximum elements of the SmartArray object.

17. Write a C++ program that generates a random integer **n** in the range [10, 25], constructs an empty SmartArray, and then appends **n** random integers in the range [5, 25] to the SmartArray object. Finally print the elements of the SmartArray object that are prime numbers.

18. **[Challenge Optional]** Consider the class declaration

```
class SmartMatrix
{
private:
        int rows;
        SmartArray *M;
public:
        ⋮
};
```

This class represents matrix objects. Give necessary constructors, destructor, setters, getters and overloaded operators in order to use this class to perform common matrix operations. In particular please provide

- Matrix addition, subtraction and multiplication operators
- Streaming in and streaming out operators
- The indexing operator [ ]. In this operator, you must return by reference so that the double indexing **A[i][j]** will be possible as discussed below.

Note that once you have a matrix object **A** with the indexing operator implemented, then you have got automatically the ability to access elements of the matrix **A[i][j]**. This is because **A[i]** will be executed first returning a **SmartArray** object; and then the **[j]** will be applied to the returned **SmartArray** object **which will access the element at index j of the SmartArray object**.

- **[Challenge]** Only if you are familiar with Linear Algebra! Given a matrix, its inverse (if it exists) is obtained by Gauss-Jordan Elimination method. Provide a member function named **getInverse** that doesn't modify the calling object and returns the inverse of the calling object.
- **[Challenge]** Only if you are familiar with Linear Algebra! Given A set of simultaneous linear equations **AX = Y**, the solution of these simultaneous linear equations (if there is one) is given by $\mathbf{X=A^{-1}Y}$. Provide a member function named **solveSimLinEq** that takes a **SmartArray** object as an argument and returns a **SmartArray** object which is a solution of the simultaneous linear equations given by **AX = Y** where **A** is the calling object.