Exercice 2 :

On considère la classe Liste qui implémente une liste simple d'éléments de type T (double) :

```
#define CAPACITE_DEFAUT 5
#define CAPACITE_PLUS 2
typedef double T;

class Liste
{
    T* data;            // tableau contenant les éléments de la liste
    int capacite;       // taille du tableau data
    int nbelement;      // nombre des éléments de la liste
public:
    ...
};
```

Définir pour cette classe, en donnant l'interface et le corps, les fonctions suivantes :

- Un constructeur qui permet de créer une liste vide, dont la capacité est fournie en argument et qui aura **CAPACITE_DEFAUT** comme valeur par défaut.
- Un constructeur qui permet de créer une liste à partir d'un tableau d'éléments de type T.
- Un constructeur par recopie (s'il est nécessaire).
- Un destructeur (s'il est nécessaire).
- Une surcharge de l'opérateur d'affectation.
- Une surcharge de l'opérateur [ ] qui permet l'accès en lecture et en écriture à un élément de la liste (sans contrôle d'indice).
- Une surcharge de l'opérateur % qui permet de tester l'appartenance d'un élément à la liste :

```
e % lst              // vaut true si e appartient à lst, false sinon.
```

- Une surcharge de l'opérateur << qui permet d'ajouter un élément à la fin de la liste, comme suit :

```
lst << e1 << e2      // ajoute e1 puis e2 à la fin de la liste lst.
```

Et une surcharge de l'opérateur >> qui permet d'ajouter un élément au début de la liste, comme suit :

```
e >> lst             // ajoute e au début de la liste lst.
```

Au cas où la liste est pleine, sa capacité sera augmentée de **CAPACITE_PLUS.**

- Une surcharge de l'opérateur << qui permet récupérer le premier élément et de le supprimer de la liste :

```
e << lst      // retire de lst son premier élément et le place dans e.
```

Et une surcharge de l'opérateur >> qui permet de récupérer les derniers éléments d'une liste en les supprimant :

```
lst >> e1 >> e2      // retire le dernier élément de lst dans e1
                     // puis retire l'avant dernier et le place dans e2.
```

- Une surcharge de l'opérateur << qui permet d'afficher les éléments d'une liste sans la modifier.

UNIVERSITE IBN TOFAIL
Faculté des Sciences
Département de Mathématiques et d'Informatique
KENITRA

Année universitaire 2019-2020

Examen – 2019/2020
SMI – POO en C++

Exercice 1 :

I. On considère la classe `Point` suivante :

```
typedef int T;
class Point{
    T x, y;
public:
    Point(T x, T y);
    ...
};
```

1. Ecrire l'interface (`point.h`) et le corps (`point.cpp`) de cette classe, en définissant :
   - Le constructeur cité sans changer les noms des paramètres et sans donner de valeur par défaut aux paramètres.
   - Une surcharge de l'opérateur ++ qui permet d'ajouter la valeur 1 à toutes les composantes d'un point.
   - Une surcharge de l'opérateur de comparaison == qui permet de comparer deux points.
   - Les fonctions nécessaires pour que le programme test suivant soit exact :

```
#include <iostream>
#include "point.h"
using namespace std;

int main()
{
    Point a(1, 2);
    const Point b (-1, 2);
    cout << 2 << " * " << a << " = " << 2*a << endl;
    cout << a << " + " << b << " = " << a+b << endl;
    cout << a << " - " << b << " = " << a-b << endl;
    return 0;
}
```

2. Peut-on déduire l'opérateur de comparaison != (entre deux points) de l'opérateur == qui est déjà défini. Sinon définissez cet opérateur.

II. On considère la classe `Pixel` qui représente un point avec une couleur :

```
class Pixel{
    Point p;
    unsigned int couleur;
public :
    Pixel(T a, T b, int c);
    Pixel(Point u, int c);
    ...
};
```

Donner l'interface et le corps de cette classe, en définissant :
- Les deux constructeurs cités de cette classe.
- Une surcharger l'opérateur de comparaison == de deux pixels.
- Une surcharge de l'opérateur << qui permet d'afficher les coordonnées et la couleur d'un pixel.

III. Peut-on écrire la classe `Pixel` comme classe dérivée de la classe `Point` ?
Si oui, définissez cette classe dérivée en réécrivant les mêmes fonctions (membres et amies) définies dans la partie II : *les deux constructeurs, et les surcharges des opérateurs == et <<.*

N.B. la classe `Point` ne devrait pas être modifiée.