Opérateurs

Ce TP est divisé en deux phases qui doivent, dans la mesure du possible, être traitées simultanément. La première phase consiste en l'implémentation pure de la classe et de ses méthodes. La seconde phase s'attachera à l'analyse et à l'amélioration du code.

L'objectif de ce second TP est de travailler sur la surcharge d'opérateurs pour la classe `Dvector`. Pour ce faire, vous reprendrez comme base de travail la classe implémentée lors de la première séance pratique.

Phase 1 : Implémentation

1. Implémenter l'opérateur d'accession `()` à un élément du vecteur. Cet opérateur prendra en paramètre des entiers compris entre `0` et `size()-1`. Cet opérateur devra permettre un accès en lecture et en écriture lorsque cela a un sens.

Relancer `verifier.py` régulièrement (pour chaque question), afin de vérifier que vous respectez bien l'énoncé. **Enfin, le relancer sur l'archive compressée (.tar.gz) avant de l'envoyer.**

2. Implémentation les opérateurs standards :
   - Implémenter l'addition, la soustraction, la multiplication et la division par un réel.
   - Implémenter l'addition et la soustraction de deux vecteurs.
   - Implémenter l'opérateur unaire $-$.
3. Implémenter les opérateurs << et >> :
   - Tous les flux `istream` n'étant pas "rembobinables", l'opérateur >> ne redimensionnera pas le vecteur destination.
   - Les opérateurs << et >> doivent être cohérents (>> doit permettre de recréer un ou plusieurs objets redirigé(s) vers le flux avec <<).
4. Implémenter les opérateurs $+=, -=, *=, /=$ lorsque l'opérande droit est un réel ou un vecteur si cela a un sens.
5. Surcharger l'opérateur d'affectation $=$. On proposera au moins deux implémentations différentes dont l'une d'elle utilisera la fonction `memcpy`.
6. Surcharger l'opérateur de test d'égalité $==$.
7. Implémenter une méthode `resize` qui permettra de changer la taille du vecteur. Elle ne fera une allocation de mémoire que lorsque c'est nécessaire. Elle prendra en premier paramètre la nouvelle taille du vecteur et en second paramètre la valeur des nouveaux éléments ajoutés si la nouvelle taille est supérieure à la précédente.

Phase 2 : Analyse

Question 1

Pour chacune des méthodes précédentes, écrire un programme test.

Question 2

Mettre en évidence la différence entre l'utilisation des 2 opérateurs + suivants :

```
Dvector operator+ (Dvector a, Dvector b)
Dvector operator+ (const Dvector &a, const Dvector &b)
```

Comparer le nombre d'objets créés dans les 2 cas.

Remarque

Pour tous les opérateurs définis :

- Bien déclarer `const` ceux qui ne modifient pas l'objet
- Bien déclarer `const` les paramètres non modifiés
- Bien renvoyer la référence vers l'objet courant ou vers un paramètre quand c'est pertinent (opérateurs "chaînables")

Lors de l'implémentation de nouveaux opérateurs, il peut être judicieux de factoriser du code (en utilisant un opérateur pour l'implémentation d'un autre) plutôt que de le dupliquer.

Cette factorisation peut réduire les performances (copies supplémentaires d'objets par exemple) mais améliorer la maintenabilité du code. Par exemple, utiliser l'opérateur d'affectation = depuis le constructeur par recopie permet d'éviter de dupliquer le code gérant l'initialisation des attributs.

Question 3

Identifier d'autres factorisations possibles.

Question 4

Comparer les performances de l'opérateur = si on utilise ou non la fonction `memcpy` pour recopier les données.

Tester l'efficacité de ces 2 versions sur des vecteurs de plusieurs millions d'éléments à l'aide de la fonction `time`. En particulier, on mesurera le temps nécessaire à la recopie sans utiliser `memcpy` et lorsque l'opérateur `()` vérifie le non dépassement des bornes.

Remarque

Utiliser `valgrind` pour vérifier que toute la mémoire a été libérée.