TP n° 2 : 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 du premier TP.

Phase 1 : Implémentation

1. * Au vu du retour que vous avez eu sur le premier TP, vérifier le bon fonctionnement :
   - Des deux constructeurs de la question 1.
   - Du constructeur par recopie.
   - Du destructeur.
   - De l'opérateur `size() const`.
   - De l'opérateur `fillRandomly()`.
   - De l'opérateur `display( std::ostream& str ) const`.
2. * 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.

Une fois les questions 1 et 2 traitées, vérifier que la structure (dossiers, fichiers, nom des classes et fonctions) répond bien aux instructions données : Lancer le script `verifier.py`, celui-ci doit confirmer que les questions 1 et 2 ont été traitées correctement. 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.**

3. 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 $-$.
4. 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 <<).
5. Implémenter les opérateurs $+=, -=, *=, /=$ lorsque l'opérande droit est un réel ou un vecteur si cela a un sens.
6. Surcharger l'opérateur d'affectation $=$. On proposera au moins deux implémentations différentes dont l'une d'elle utilisera la fonction `memcpy`.

7. Surcharger l'opérateur de test d'égalité $==$.
8. Implémenter une méthode `view (bool copy, int start, int count)` permettant de visualiser (et manipuler) une sous-partie d'un vecteur (un bloc) comme un vecteur à part entière.
   - Cette fonction `view` créera donc un vecteur et prendra en argument 2 entiers $start$ et $count$. Le premier est le premier indice (compris entre `0` et `size()-1`) du bloc à considérer et le second, le nombre d'éléments. Dans un souci d'efficacité, la fonction `view` ne procédera pas à une recopie des données sauf si `copy==true`. Pour ce faire, ajouter à la classe `Dvector` un attribut permettant de savoir si l'instance de la classe est propriétaire ou non de son pointeur de données (cet attribut doit être à `true` pour tous les objets créés autrement qu'avec la fonction `view`). Modifiez en conséquence l'ensemble des méthodes précédemment écrites.
   - Notez que "non-propriétaire" ne signifie pas "constant". Un objet qui n'est pas propriétaire de ses données peut tout de même les modifier.
   - Pour permettre le test du bon fonctionnement de cette méthode :
     - Implémenter une méthode `bool isOwner() const` renvoyant l'attribut permettant de savoir si l'instance de la classe est propriétaire ou non de son pointeur de données.
     - Implémenter une méthode `const double* getData() const` renvoyant le pointeur de données `double*` de l'objet `Dvector`.
   - Dans le cas de l'opération = entre les vecteurs `a` et `b`, on distingue les cas suivants :

     Si `a` et `b` ont la même taille

	a	b	a, après a=b
cas 1	P	P	P
cas 2	P	NP	P
cas 3	NP	P	NP
cas 4	NP	NP	NP
cas 5	0	P	P
cas 6	0	NP	NP

     Si `a` et `b` ont des tailles différentes

	a	b	a, après a=b
cas 1	P	P	P
cas 2	P	NP	P
cas 3	NP	P	Err
cas 4	NP	NP	Err
cas 5	0	P	P
cas 6	0	NP	NP

     où P signifie que le vecteur est propriétaire de ses données, NP que le vecteur n'est pas propriétaire de ses données, Err signifie que l'opération doit lever une exception (utilisez `throw`) et 0 signifie que le vecteur n'a pas été initialisé (vecteur de taille 0). La troisième colonne de chacune des tables donne l'état de la variable `a` après l'opération =.
   - Dans le cas du constructeur par recopie, l'état "P" ou "NP" doit évidemment être préservé.
   - La méthode view appelée sur un `Dvector` de taille nulle devra lever une exception (utilisez `throw`). Idem si *count* vaut 0. En fonction des paramètres de la méthode, identifiez tous les autres cas pour lesquels une exception doit être levée.

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.

Question 3. *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")

Question 4. ** 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'initailisation des attributs (rendant ainsi plus facile l'évolution du code, pour traiter la question 8 par exemple).*
Identifier d'autres factorisations possibles. Que vous décidiez de les implémenter (meilleure maintenabilité) ou pas (meilleures performances), les mentionner dans le rapport (exemple : On peut utiliser l'opérateur ... pour l'implémentation de l'opérateur ...).

Question 5. ** 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.*

Question 6. *Utiliser valgrind pour vérifier que toute la mémoire a été libérée.*