Programmation orientée objet en C++

Travaux pratiques séance 4

A la fin du TP, déposez une archive avec votre travail (sources + compte-rendu concis) sur Dokeos (formation.u-psud.fr), avec le sujet [443]"nom"-"prenom"-travail-TP4 (par exemple [443]Dupont-Pierre-travail-TP4). Avant minuit, déposez également sur Dokeos le compte-rendu détaillé (CR) correspondant à votre travail ; les modifications dans le code ultérieures à la fin du TP ne sont pas prises en compte - concentrez vous sur la rédaction du CR.
Conseils de base :

- le TP est **individuel** ; toute fraude sera sanctionnée drastiquement.
- dès que possible (toutes les quelques lignes idéalement), **compilez** et **testez** votre code. L'erreur la plus fréquente des débutants est d'écrire des dizaines de lignes de code sans tester, et de se retrouver tout à coup avec plein d'erreurs de compilation et de bugs
- si vous dupliquez du code dans vos méthodes, cela veut dire qu'il y a probablement un problème : soit une méthode devrait appeler l'autre, soit le code commun devrait être mis dans une méthode auxiliaire, qui est appelée chaque fois que sa fonctionnalité est nécessaire
- même si cela ne vous rapporte pas une meilleure note tout de suite, essayez d'aller au bout de ce sujet avant le TP suivant. Cela vous améliorera sûrement votre niveau, ainsi que les notes suivantes.

Des éléments pris en compte lors de la notation :

- vous avez respecté ce qu'on demande (la date limite pour l'envoi de votre code et compte-rendu, le format de la soumission etc.)
- votre code contient des **commentaires** qui justifient bien le rôle de chaque classe, membre de classe, méthode, et qui expliquent les détails moins évidents d'implémentation
- le compte-rendu est un document soigné et bien organisé (introduction, rappel bref de l'objectif, contenu, discussion finale) qui justifie tous vos **choix** de design des classes (i.e. pour quoi ce membre est initialisé par défaut ainsi, pour quoi l'allocation de ce tableau se fait dans cette méthode et pas dans le constructeur etc.), mais on ne met ni les détails d'implémentation (pour cela on a les commentaires), ni des copier-coller des classes et des méthodes entières pour faire du volume
- la syntaxe de votre programme est **correcte** (i.e. le code compile)
- votre programme fonctionne correctement (i.e. le résultat correspond à ce qui est demandé), les tests demandés sont implémentés
- votre code est de bonne qualité (i.e. pas de fuites de mémoire, listes d'initialisation pour les constructeurs, encapsulation efficace, utilisation correcte des attributs vus en cours (public, private, const, static etc.) bonne organisation du code en méthodes)

1 Gestion d'une mémoire tampon pour un ensemble de capteurs

Le but de ce TP est de réaliser une mémoire tampon (un buffer) pour sauvegarder des données qui arrivent de plusieurs capteurs, jusqu'au moment où un autre processus de traitement demande ces données. La communication autour du buffer est encodée dans un fichier text comme sensors.txt :

```
0 3 4
0 1 5
0 2 4
0 3 6
0 1 7
1 2
1 4
```

1 1
0 1 10

La signification des valeurs ligne par ligne est la suivante :
- réception (0) de la part du capteur 3 de la valeur 4
- réception (0) de la part du capteur 1 de la valeur 5
- réception (0) de la part du capteur 2 de la valeur 4
- réception (0) de la part du capteur 3 de la valeur 6
- réception (0) de la part du capteur 1 de la valeur 7
- demande (1) de la valeur la plus ancienne reçue de la part du capteur 2
- demande (1) de la valeur la plus ancienne reçue de la part du capteur 4
- demande (1) de la valeur la plus ancienne reçue de la part du capteur 1
- réception (0) de la part du capteur 1 de la valeur 10

Le tampon doit garder **en ordre chronologique** les données reçues et fournir aux processus de traitement les données en ordre d'arrivée (donc il s'agit de gérer une file d'attente classique de type FIFO, first in - first out). Par exemple, pour la première demande de l'exemple, le buffer retire de la file la valeur 4 qu'il avait reçue de la part du capteur 2. Par la suite il ne fait rien, puisqu'il ne peut pas trouver dans la file aucune donnée de la part du capteur 4. Enfin, il retire de la file la valeur 5 qu'il avait reçue de la part du capteur 1. A la fin de ces instructions la file contient donc en ordre (en notation indice capteur - valeur) :

3 4
3 6
1 7
1 10

Exercice 1 — Une classe qui sauvegarde une donnée instantanée

Écrivez une classe template `template <class T> class SensorData` qui contient un champ entier, l'identifiant du capteur qui a généré la donnée respective, et un champ générique (template) de type T qui représente la valeur envoyée par le capteur. Il faut écrire un constructeur à deux paramètres, un constructeur de copie, l'opérateur = et deux assesseurs publiques pour les champs prives spécifiés ci-haut.

Exercice 2 — Un conteneur de base

Écrire une classe template `template <class T> class BaseSensorBuffer` qui gère les données qui arrivent des capteurs. Pour cette implémentation de base, qui est la plus simple possible, on n'utilise pas de tampon (on perd les données). On commence par implémenter une méthode publique `void read(const char* fName, const char* outputFile)`, qui ouvre en mode lecture un fichier en utilisant un objet `fstream`. Le fichier est indiqué par le premier paramètre (par exemple sensors.txt), et on lit toutes les valeurs en boucle jusqu'à la fin du fichier. Vous pouvez consulter le lien

http://stackoverflow.com/questions/21647/reading-from-text-file-until-eof-repeats-last-line

pour éviter quelques erreurs d'utilisation.
Si on lit un 0 (arrivée des données) on sait ce que cela représente les deux valeurs suivantes, on construit un objet de type `SensorData` et on appelle une méthode **privée virtuelle** qui a la signature `virtual void store(SensorData<T>& data)` qui devrait stocker la donnée respective dans un conteneur. Dans le cas de cette classe de base, on ne stocke rien ; on se contente juste d'afficher à l'écran un message de type :

```
storing data 4 from sensor 3
```

Si on lit un 1 (requête des données) on sait ce que cela représente la valeur suivante, et on appelle une méthode **privee virtuelle** qui a la signature `virtual void retrieve(int sensorIdx)` qui devrait chercher la donnée la plus ancienne correspondant au capteur respectif dans le conteneur. Dans le cas de cette classe de base, on ne cherche rien ; on se contente juste d'afficher à l'écran un message de type :

```
retrieving data from sensor 2
```

Enfin, une fois que la méthode `read` arrive à la fin du fichier d'entrée, elle doit appeler une méthode privée `void saveBufferToFile(const char* outName)` qui va sauvegarder dans un fichier texte de sortie

la configuration du conteneur à la fin de l'exécution de toutes les opérations demandées dans le fichier d'entrée. Dans le cas de cette classe de base, on se contente juste d'écrire dans ce fichier :

```
I am just a base buffer, I did not save any data actually. Sorry, human
```

A la fin, vous devriez pouvoir appeler votre classe à partir du fichier principal de la manière suivante :

```
BaseSensorBuffer<int> bsb;
bsb.read("sensors.txt", "outputBase.txt");
```

Exercice 3 — Conteneur sous la forme d'un vecteur

Déclarer une classe `template <class T> class VectorSensorBuffer` qui hérite de la classe de base précédente, et qui a un conteneur privé sous la forme d'un objet vector STL :

```
vector < SensorData<T> > buffer;
```

Re-implémentez les méthodes virtuelles de la classe de base, c'est à dire `store, retrieve` et `saveBufferToFile` pour utiliser effectivement le conteneur de type vector pour insérer et enlever des valeurs du buffer. On vous recommande de faire l'insertion des valeurs qui arrivent à la fin du vecteur (donc la fin du vecteur sera la fin de la file).
Votre classe devrait pouvoir s'utiliser de la manière suivante :

```
VectorSensorBuffer<int> vsb;
vsb.read("sensors.txt", "outputVector.txt");
```

Attention : la méthode `read` de la classe de base n'a pas besoin de changer.
Vérifiez bien, en regardant dans le fichier `outputVector.txt` que la configuration finale du conteneur est correcte.
Utilisez bien la doc en ligne de STL, y compris les exemples pour les itérateurs. Pour déclarer un itérateur sur un vecteur d'objets template, regardez également la page suivante :

http://stackoverflow.com/questions/3311633/nested-templates-with-dependent-scope

Exercice 4 — Conteneur sous la forme d'une liste

Implémentez un conteneur avec les mêmes objectifs que pour le vecteur, en utilisant cette fois une list de STL.
Votre classe devrait pouvoir s'utiliser de la manière suivante :

```
ListSensorBuffer<int> lsb;
lsb.read("sensors.txt", "outputList.txt");
```

Vérifiez que la configuration de sortie est identique à la précédente.

Exercice 5 — Complexité des conteneurs

Les conteneurs fonctionnent de manière identique, et pourtant la complexité des calculs n'est pas forcement la même. Pour tester l'efficacité de chaque classe, vous avez aussi à votre disposition un fichier d'instructions `largedata.txt` de taille beaucoup plus grande (100000 instructions, 2048 capteurs, en moyenne 9 arrivées des données capteurs pour une requête de données). Exécutez vos classes sur ce fichier d'entrée, ce qui devrait vous montrer qu'il y a une différence de complexité assez importante entre les deux conteneurs. En regardant la doc de STL, expliquez quelles sont les opérations susceptibles d'expliquer cette différence.
Attention : dans ce cas, les valeurs envoyées par les capteurs sont de type float, donc n'oubliez pas de changer le type de données.

Exercice 6 — Reflexions algorithmiques

Si vous avez compris quelles sont les opérations qui coûtent cher pour les implémentations précédentes, essayez d'implémenter un conteneur en utilisant les structures STL les plus adaptées qui soit le plus rapide possible pour la conception de ce buffer.