ISIMA 3ème année - MODL/C++
TP 5 : Bibliothèque standard

On souhaite réaliser une application permettant d'analyser des données statistiques : on considère un échantillon de valeurs et l'on veut obtenir un histogramme qui répartit les valeurs dans des « classes » (au sens statistique, https://fr.wikipedia.org/wiki/Histogramme). Essayez d'utiliser au maximum les fonctionnalités de la bibliothèque standard.

1) Nous allons définir une classe `Echantillon` qui regroupe des valeurs (classe `Valeur`).

   Définir d'abord la classe `Valeur`, on stockera dans un premier temps un simple réel. Tests 1-4

   Définir ensuite la classe `Echantillon`, un simple vecteur suffira pour stocker les valeurs. Dans cette classe, proposer des méthodes pour ajouter des valeurs, obtenir les valeurs minimale et maximale, et accéder à n'importe quelle valeur à partir de son indice. Gérer les erreurs éventuelles en levant des exceptions (utiliser les classes standards). Tests 5-9

   Indications : Pour les valeurs minimale et maximale, utiliser les algorithmes `std::min_element` et `std::max_element` de la bibliothèque standard, avec une lambda comme politique passée en argument pour comparer les valeurs. Les alternatives peuvent être de proposer une surcharge de l'opérateur `<` pour les valeurs (car les algorithmes l'utilisent ici par défaut) ou un foncteur, ce qui éviterait la répétition du code de comparaison constaté en utilisant les lambdas.

2) Nous allons définir une classe `Histogramme` qui contient un ensemble de « classes » (au sens statistique), un vecteur suffira initialement.

   Définir la classe `Classe` représentant une « classe », c'est-à-dire un intervalle (une borne inférieure et une borne supérieure) avec une quantité. Proposer une méthode `ajouter` qui incrémente simplement la quantité. Tests 10-11

   Définir la classe `Histogramme`. L'intervalle de valeurs (bornes inférieure et supérieure) de l'histogramme, ainsi que le nombre de classes seront fournis à la construction, afin de générer automatiquement les classes couvrant l'intervalle de valeurs avec toutes la même amplitude.
   Test 12

3) Compléter la classe `Histogramme` avec une méthode `ajouter` qui remplit un histogramme à partir d'un échantillon. Test 13

   Indications : Ecrire d'abord une méthode `ajouter` qui ajoute une simple valeur dans l'histogramme, l'utiliser ensuite pour écrire la méthode d'ajout d'un échantillon. Pour déterminer dans quelle classe se trouve une valeur, utiliser de préférence l'algorithme `std::find_if`, avec une lambda comme prédicat.

On souhaite maintenant pouvoir obtenir une vision alternative de l'histogramme dans laquelle les « classes » sont présentées par ordre décroissant de quantité (ainsi la première classe sera celle contenant la plus grande quantité de valeurs de l'échantillon, la dernière étant celle contenant la plus petite quantité).

4) On propose l'approche suivante : la classe `Histogramme` utilise un ensemble (`std::set`) plutôt qu'un vecteur pour stocker les classes. Par défaut, l'ensemble est trié en fonction du résultat de l'opérateur < sur les éléments stockés. Définir l'opérateur < pour les classes de manière à ce qu'elles soient rangées par ordre croissant de leur borne inférieure. Test 12

   Indications : Pour modifier un élément stocké dans un ensemble, il faut le retirer et ensuite le remettre, car le tri se fait à l'insertion (impact sur l'ajout d'une valeur dans l'histogramme).

5) Modifier la classe `Histogramme` pour que l'on puisse choisir l'ordre dans lequel ranger les classes. Par défaut, l'ensemble utilise le foncteur `std::less` de la bibliothèque standard pour trier les éléments (qui fait lui même appel à l'opérateur < des éléments), mais il est possible de définir son propre foncteur (opérateur binaire qui compare deux éléments) pour trier les classes par quantité décroissante notamment.

   La classe devient donc générique avec comme paramètre le type du foncteur qui va servir à trier les classes. Mettre par défaut le foncteur `std::less` pour ce paramètre, afin que les classes soient triées par ordre croissant. Test 14

   Essayer ensuite cette nouvelle classe avec le foncteur `std::greater` (nécessite l'opérateur > pour les classes). Test 15

   Et enfin avec un foncteur *ad hoc* (`ComparateurQuantite` à écrire) permettant un tri des classes par quantité décroissante (si deux classes ont la même quantité, l'ordre des bornes inférieures est appliqué). Test 16

6) On ne peut pas changer dynamiquement de relation d'ordre sur les classes. Proposer un constructeur (*template*) pour convertir un histogramme avec une certaine relation d'ordre en un histogramme avec une autre relation d'ordre (s'inspirer de la copie de piles vue en cours). Test 17

Dans notre modélisation, nous avons oublié que les nombres sont des notes associées à des étudiants. Nous allons à présent réparer cet oubli.

7) Modifier la classe `Valeur` pour y intégrer un étudiant (son nom) et sa note (`getNombre` retourne la note). Tests 18-21

8) Ajouter une structure dans l'histogramme permettant de conserver les valeurs se trouvant dans chacune des classes. Les classes `Valeur` et `Classe` n'ont pas à être modifiées, on propose d'utiliser une structure associative à clé multiple (`std::multimap`) de manière à pouvoir associer plusieurs valeurs à une classe. Test 22

9) Ecrire une méthode `getValeurs` dans la classe `Histogramme` qui permet d'obtenir la liste de toutes les valeurs d'une « classe ». Il s'agit d'identifier les valeurs associées à la classe dans la *multimap* définie précédemment (utiliser `std::equal_range`). Test 23

10) Surcharger l'opérateur << pour la classe `Histogramme`, afin d'afficher les classes en fonction de l'ordre défini dans l'histogramme, et pour chaque classe, la liste des étudiants avec leur note. Tester avec un échantillon rempli aléatoirement pour obtenir par exemple la sortie suivante (https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution) :

```
Echantillon = 3.03162 10.6985 1.09399 10.748 13.6283 10.281 6.56962 2.34028 6.70708 18.267 10.2371
5.42167 6.14 5.29529 19.6752 5.72411 8.03705 7.5594 17.4811 15.6231

Histogramme tri par defaut:
[0;2] = 1 : (inconnu;1.09399)
[2;4] = 2 : (inconnu;3.03162) (inconnu;2.34028)
[4;6] = 3 : (inconnu;5.42167) (inconnu;5.29529) (inconnu;5.72411)
[6;8] = 4 : (inconnu;6.56962) (inconnu;6.70708) (inconnu;6.14) (inconnu;7.5594)
[8;10] = 1 : (inconnu;8.03705)
[10;12] = 4 : (inconnu;10.6985) (inconnu;10.748) (inconnu;10.281) (inconnu;10.2371)
[12;14] = 1 : (inconnu;13.6283)
[14;16] = 1 : (inconnu;15.6231)
[16;18] = 1 : (inconnu;17.4811)
[18;20] = 2 : (inconnu;18.267) (inconnu;19.6752)

Histogramme tri par quantite:
[6;8] = 4 : (inconnu;6.56962) (inconnu;6.70708) (inconnu;6.14) (inconnu;7.5594)
[10;12] = 4 : (inconnu;10.6985) (inconnu;10.748) (inconnu;10.281) (inconnu;10.2371)
[4;6] = 3 : (inconnu;5.42167) (inconnu;5.29529) (inconnu;5.72411)
[2;4] = 2 : (inconnu;3.03162) (inconnu;2.34028)
[18;20] = 2 : (inconnu;18.267) (inconnu;19.6752)
[0;2] = 1 : (inconnu;1.09399)
[8;10] = 1 : (inconnu;8.03705)
[12;14] = 1 : (inconnu;13.6283)
[14;16] = 1 : (inconnu;15.6231)
[16;18] = 1 : (inconnu;17.4811)
```

Indications : Utiliser de préférence l'algorithme `std::generate_n` pour remplir l'échantillon avec des valeurs aléatoires. Pour cela, utiliser `std::back_inserter` pour obtenir un *output iterator* sur l'échantillon à fournir à l'algorithme, l'itérateur appellera la méthode `push_back` de l'échantillon à chaque déréférencement. Votre classe `Echantillon` devra donc disposer d'une interface minimale issue des conteneurs standards, à savoir le type interne `value_type` et la méthode `push_back`.