Génie Logiciel TP4 : généricité sur une classe

Dans ce TP, on veut illustrer l'utilisation de la généricité pour la définition d'une classe représentant un point d'un espace Euclidien de type et de dimension quelconque. Nous allons partir d'un code spécifique aux points en 3D avec coordonnées réelles représentées par des **`double`** et nous allons rendre progressivement ce code **générique** en remplaçant certains types ou valeurs par des **paramètres template**.

Pour des raisons techniques [1], l'ensemble de la classe sera définie dans **un seul fichier d'en-tête**. À cet effet, on vous propose **deux versions du code original** : une première (suffixée *inline*) dans laquelle la classe est déclarée et définie dans un même bloc, et une seconde (suffixée *out-of-line*) dans laquelle la classe est d'abord déclarée puis définie quelques lignes en-dessous, comme si on avait concaténé les fichiers en-tête et source utilisés jusque-là. **Choisissez** celle que vous voulez et importez-la depuis `main.cpp`.

Le code fourni est complet et doit donc pouvoir être compilé et exécuté avant de commencer le TP.

1 Paramètre template pour un type

La première étape consiste à **rendre générique le type** de représentation des coordonnées (ici **`double`**) de telle sorte à pouvoir utiliser une représentation plus précise (par exemple **`long double`**) ou moins précise (par exemple **`float`**).

Cette modification consiste, en général, à suivre les étapes suivantes :

1. **déclarer** un paramètre template de la classe pour un **type** (mot-clé **`typename`** ou **`class`** dans la déclaration **`template`**) et à lui donner un nom (en général `T`),
2. **remplacer** toutes les occurrences du type original par ce nom.

Pensez à modifier également la surcharge de l'opérateur `<<` (paramètre template pour une fonction), à adapter l'instanciation du `Point` dans `main.cpp`, et à **tester** le tout avec les différents types proposés au-dessus.

2 Paramètre template pour un nombre

On peut également utiliser un **paramètre template pour spécifier un nombre** dont dépend la définition de la classe, par exemple un entier qui correspondrait ici à la **dimension de l'espace** auquel appartient le point.

Ce genre de paramètre template se déclare en indiquant le type de l'entier à la place du mot-clé **`typename`** (ou **`class`**) et en lui associant un nom, par exemple :

```
1 template <std::size_t N> // ou <int N> si vous préférez
2 class Point
3 { ...
```

En se rappelant que l'on peut spécifier **plusieurs paramètres template**, séparés par des virgules, ajoutez un paramètre template pour pouvoir instancier des `Point` de **dimension arbitraire**, adaptez le reste du code et **testez** avec différentes dimensions.

1. La classe générique ne peut être compilée qu'à l'endroit où les templates sont spécifiés, typiquement à l'instanciation de la classe, sauf si toutes les combinaisons de template sont connues à l'avance ce qui peut permettre d'instancier les templates dans un fichier source dédié.

3 Spécialisation partielle des paramètres template

On remarque par ailleurs qu'en **dimension 1** le calcul de la **norme** du vecteur peut être grandement simplifié (et accéléré) en ne renvoyant que la **valeur absolue** de la coordonnée plutôt que la racine carrée du carré.

On souhaiterait donc proposer une **spécialisation partielle** de la classe `Point` pour la dimension 1 tout en restant générique par rapport au type de représentation des coordonnées [2].

Une spécialisation partielle des paramètres template consiste donc à **dupliquer** entièrement le code de la classe `Point` (voir la note de pied de page pour des solutions plus économiques) en ne laissant plus qu'un seul paramètre template (la représentation des coordonnées) et en indiquant la spécialisation choisie juste après le nom de la classe.

En pratique, on va donc avoir **deux implémentations** de la classe `Point` et le code sera structuré comme suit :

```
1  template <typename T, std::size_t N>
2  class Point
3  {
4      // Version générique en type et dimension
5      ...
6  };
7
8  template <typename T> // Il n'y a plus qu'un seul paramètre template ...
9  class Point<T, 1>     // ... mais on spécifie les deux apres Point.
10 {
11     // Version spécialisée pour la dimension 1
12     ...
13 };
```

Implémentez cette spécialisation et **testez**-là.

4 Bonus : occupation mémoire

Le code fourni affiche également l'occupation mémoire d'une instance `Point` en fonction des paramètres template choisis, par exemple 24 octets en dimension 3 et en représentant les coordonnées par des **`double`**, ou 8 octets en dimension 2 et en utilisant des **`float`**.

Pour ces mêmes cas, quelle serait l'**occupation mémoire** d'un point si on avait implémenté la classe `Point` de telle sorte que la dimension de l'espace soit choisie **dynamiquement**, c'est-à-dire en la spécifiant en paramètre du constructeur plutôt que via un paramètre template, et donc en utilisant une allocation dynamique (**`new`**/**`delete`**) plutôt qu'un tableau statique ?

2. C'est en réalité probablement inutile ici : un simple bloc **`if`** dans la méthode `norme` suffirait et le compilateur optimiserait le branchement vu qu'il connaît la dimension à la compilation. Sans cela, il existe également des moyens détournés pour spécialiser partiellement des méthodes sans réécrire toute la classe ou des syntaxes plus récentes comme les blocs **`if`** `constexpr` introduits dans la norme C++17.