Programmation Orientée Objet - C++ - TP 8
*Découverte des templates*

---

### Exercice 1 : Merci Patron !

Les templates permettent d'écrire du code générique (sans spécifier de type) que le compilateur utilisera pour générer le code en fonction des types nécessaires.

Soit la fonction template `getMinimum` :

```
template<typename T>
const T &getMinimum(const T &a, const T &b)
{
    return a < b ? a : b;
    // équivaut à :
    // if (a < b) return a;
    // else return b;
}
```

« Mais c'est quoi cette syntaxe avec un point d'interrogation ? »

Il s'agit d'une affectation ternaire... Je vous mets ça là pour que vous connaissiez,

mais évitez de l'utiliser à tout bout de champs (en fait, ne l'utilisez pas :-) ) !

« Il retourne une référence constante ? »

Et pourquoi pas ? !

**Q.1.** Recopiez cette fonction dans votre code et testez-la en l'appelant avec :
- deux entiers (*e.g* 79 et 67) ;
- deux flottants (*e.g* 7.9 et 6.7) ;
- deux chars (*e.g* 'a' et 'z').

Normalement tout se passe bien, le compilateur a généré le code pour chaque type demandé.

**Q.2.** Maintenant, essayez d'appeler cette fonction avec un flottant et un entier (*e.g* 7.9 et 67). Que se passe-t-il ? Proposez une solution pour résoudre ce problème.

**Q.3.** Appelez la fonction `getMinimum` avec comme paramètres 'a' et 'Z'. Le résultat doit être 'Z'. Or, nous aimerions que la fonction retourne 'a'. Proposez une solution pour changer le comportement de la fonction pour les caractères (NB : la fonction `toupper` du fichier `cctype` permet de convertir une lettre en majuscule).

### Exercice 2 : Un patron qui a la classe

Comme pour les fonctions, il est possible de créer des classes templates.

**Q.1.** Créez une classe template `Point3D` qui contient 3 scalaires (du type donné en paramètre du template) comme attributs et qui définit les méthodes suivantes :
- constructeur par défaut qui initialise les attributs à 0
- constructeur prenant en paramètres 3 scalaires
- constructeur par copie
- opérateur d'affectation
- opérateur `+` avec un scalaire
- opérateur `+` avec un autre `Point3D`
- opérateur `<<`

Testez bien le fonctionnement de votre classe pour plusieurs types (`int`, `float`, ...).

**Q.2.** Proposez un constructeur permettant d'initialiser un `Point3D` à partir d'un `Point3D` d'un autre type, *e.g* :

```
Point3D<float> A(7.9f, 6.7f, 1.6f);
Point3D<int> B(A); // Contient [7, 6, 1]
```

Quel est le problème ? Comment le résoudre ?

### Exercice 3 : Et pourquoi pas Tron ? (je sais, c'est pas terrible)

Nous allons créer une classe template permettant de gérer un tableau dont la taille maximale est connue à la compilation.

**Q.1.** Créez une classe template `Tableau` qui permet de représenter un tableau de type `T` et de taille `N`. Ces variables sont les paramètres du template. Cette classe possède aussi deux attributs : `data`, le tableau et `size`, le nombre d'éléments courant dans le tableau.

**Q.2.** Ajoutez les méthodes suivantes :
- `isEmpty` : retourne `true` si le tableau est vide, `false` sinon
- `getSize` : retourne le nombre d'éléments courant
- `getMaxSize` : retourne le nombre d'éléments maximum
- `getFirst` : retourne une référence sur le premier élément (exception si impossible)
- `getLast` : retourne une référence sur le dernier élément (exception si impossible)
- l'opérateur `[]` pour pouvoir lire et modifier un élément du tableau (pas d'exception)
- `at` : pour pouvoir lire et modifier un élément du tableau (exception si impossible)
- `push` : insère un élément en fin de tableau (exception si impossible)
- `pop` : supprime le dernier élément (exception si impossible)
- `insert` : insère un élément à un indice donné (exception si impossible)
- `erase` : supprime l'élément de l'indice donné (exception si impossible)
- l'opérateur `<<` pour afficher le tableau tel que : `[ 0 1 2 3 4 ]`

Utilisez la classe d'exception que vous avez codé précédemment. Les exceptions doivent décrire l'erreur au mieux (un indice non valide est différent d'une capacité maximale atteinte).