# ISIMA 3ème année - MODL/C++
# TP 2 : Généricité

**Exercice 1**

On reprend la première partie du TP précédent. On suppose que les classes **Point**, **Cartésien** et **Polaire** ont été correctement définies et sont fonctionnelles, et que les moyens de conversion entre **Cartésien** et **Polaire** existent.

1) Définir la classe **Nuage**, générique sur le type de points à contenir. Utiliser la classe `std::vector` pour contenir les points.

2) Définir une fonction générique `barycentre()` prenant en argument un nuage de points et retournant le barycentre, supposé du même type que les points. Proposer une solution générique qui s'appuie sur la formule connue pour les cartésiens (utiliser des conversions).

3) Supposons que l'on connaisse la formule du barycentre en coordonnées polaires. Proposer une spécialisation pour les polaires de la fonction générique `barycentre()`.

4) On souhaite que la fonction `barycentre()` s'affranchisse aussi du type du conteneur de points. Pour cela, la classe **Nuage** doit proposer une interface similaire aux conteneurs STL, à savoir des méthodes `begin()` et `end()`, ainsi qu'un type interne `iterator`. Proposer une fonction `barycentre()` qui puisse fonctionner indifféremment sur les principaux conteneurs STL et la classe **Nuage**.

**Exercice 2**

Un calcul peut parfois être évalué en partie à la compilation (par exemple la somme de deux constantes), le reste étant évalué à l'exécution du programme. Les génériques peuvent être utilisés pour augmenter et automatiser la part de calcul évaluée à la compilation dans une expression, et de ce fait, accélérer son évaluation à l'exécution. On appelle cette technique l'évaluation partielle.

1) Factorielle et fonctions récursives

   Certains algorithmes demandent parfois de manipuler des valeurs constantes non-triviales qu'il est maladroit de cacher sous la forme d'une simple constante nommée. Par exemple, le calcul de la série de Taylor d'une fonction peut faire appel à la série de constantes *1!*, *2!*, *3!*, …, *n!*. Une solution classique revient à définir des constantes:

```
static const int FACT2 = 2;
static const int FACT3 = 6;
static const int FACT4 = 24;
static const int FACT5 = 120;
```

   Cette solution est peu pratique car elle nécessite l'ajout de constantes nommées qui vont polluer inutilement l'espace de noms global. En outre, passé certaines valeurs (*20!* par exemple), le développeur va devoir effectuer certains calculs parfois lourds avant de remplir la constante.

Réaliser une structure générique contenant un membre de classe de type unsigned long et qui permet d'effectuer le calcul de *n!* de manière statique. Au final, l'utilisation de cette structure se fera comme indiqué ci-dessous:

```
unsigned long fact20 = Factorielle<20>::value;
```

2) Calcul de série de Taylor

L'expression d'une fonction *f(x)* sous la forme d'une série de Taylor fait intervenir l'ordre *N* et la valeur du point *x* où l'on évalue la fonction. L'idée est de fournir une structure générique qui va construire à la compilation et de manière récursive la série de Taylor de la fonction à l'ordre voulu. À l'exécution, ce développement sera évalué en un point quelconque. On a donc une partie statique et une partie dynamique. Ainsi, par exemple, l'évaluation en *0* de la série de Taylor à l'ordre *5* d'exponentielle sera donné par

```
double val = Exponentielle<5>(0.0);
```

On souhaite fournir la série de Taylor des fonctions exponentielle, sinus et cosinus. On rappelle que

$$\begin{aligned}
\exp(x) &\approx \sum_{k=0}^{N} \frac{x^k}{k!} \\
\cos(x) &\approx \sum_{k=0}^{N} (-1)^k \frac{x^{2k}}{(2k)!} \\
\sin(x) &\approx \sum_{k=0}^{N} (-1)^k \frac{x^{2k+1}}{(2k+1)!}
\end{aligned}$$

Proposer les structures génériques nécessaires à ces calculs.