

ISIMA 3^{ème} année - MODL/C++

TP 3 : Généricité avancée

L'objectif ici est de concevoir une fonction chaîne qui puisse convertir n'importe quel type de valeur en une chaîne de caractères, dans le but par exemple de permettre la sérialisation (<https://en.wikipedia.org/wiki/Serialization>). Nous allons notamment prévoir la sérialisation d'un tuple (<https://en.cppreference.com/w/cpp/utility/tuple>), structure introduite dans la bibliothèque standard avec C++11.

Exercice 1 - Exceptions

Pour commencer, nous allons définir une première version de la fonction chaîne, une fonction générique avec un paramètre *template* qui est le type de la valeur à convertir en chaîne de caractères. Cette version lèvera systématiquement une exception. Des surcharges de cette fonction viendront ensuite définir des conversions pour des types spécifiques.

- 1) Ecrire la fonction générique `chaîne<T>(x)` qui devrait retourner une chaîne de caractères représentant l'objet `x` de type `T` passé comme argument.

Cette première version générique lève systématiquement une exception du type `ExceptionChaine` (à créer) pour indiquer que la conversion n'est pas possible. Cette classe d'exceptions devra hériter de `std::exception` et redéfinir la méthode `what`. Vous verrez dans les tests unitaires le format attendu pour le retour de `what` : un message suivi du type de la valeur qui n'a pas pu être convertie. Test 1

Indications : Le type de la valeur peut être obtenu sous forme de chaîne de caractères via l'instruction `typeid` vue en cours (vous appliquerez la fonction `demangle` fournie dans le fichier `demangle.hpp` à cette chaîne pour la « décoder »). Cette information sera transmise à l'exception via son constructeur. Conseil : conserver la valeur dans un attribut (pour pouvoir proposer un accesseur), et utiliser la valeur dès le constructeur pour préparer le message de retour de la méthode `what` (stocker le message dans un attribut `std::string` et retourner le tableau de caractères associé via sa méthode `c_str` dans la méthode `what`).

Question subsidiaire : Dans la classe de l'exception, utiliser un constructeur *template* prenant en argument la valeur à convertir plutôt que directement la chaîne du type obtenue avec `typeid`.

- 2) Fournir des surcharges de la fonction chaîne pour les types `string`, `int` et `double`. Pour `int`, utiliser un objet `stringstream` pour effectuer la conversion ; pour `double`, utiliser la fonction `std::to_string`. En revanche, aucune surcharge ne devra être fournie pour `long`. Test 2

Indications : L'approche avec `stringstream` fonctionne pour tout type qui possède l'opérateur `<<` pour un flux ostream (rappel : `std::cout`, `std::ofstream` et `std::stringstream` héritent de cette classe). Créer un objet `stringstream` et utiliser l'opérateur de flux pour y écrire la valeur à convertir. Appeler ensuite sa méthode `str` pour récupérer le contenu du flux sous la forme d'une chaîne de caractères.

Remarque : Pour tous les types de nombres, on peut utiliser la fonction `std::to_string` introduite en C++11 qui effectue directement la conversion.

Exercice 2 – Variadic template

- 3) Ecrire une surcharge de la fonction chaîne sous la forme d'un *template* avec un nombre variable de paramètres (cf. *variadic template*). Cette fonction doit convertir chaque argument en chaîne de caractères et les concaténer en les séparant par un espace. Test 3

Indications : Il s'agit ici de faire l'expansion du pack d'arguments de la fonction pour obtenir une séquence d'appels à la fonction chaîne pour chacun des arguments et leur concaténation. Nous avons vu brièvement l'expansion simple en cours (sous forme de liste, les arguments sont séparés par des virgules), mais il y a aussi l'expansion avec des opérateurs binaires (cf. *fold expressions*, depuis C++17). Nous vous invitons à regarder la documentation suivante pour trouver comment faire l'expansion ici : <https://en.cppreference.com/w/cpp/language/fold>. En résumé, il est possible de faire l'expansion d'un pack d'arguments sous la forme d'une suite d'opérations binaires, par exemple : l'expression `(args + ...)` est développée en `(arg1 + (arg2 + (... + argn)))`, mais attention, les parenthèses et la position des `...` ont leur importance.

Le mécanisme de paramètres variables des *templates* a permis notamment de concevoir la classe `std::tuple`, qui est une structure permettant de rassembler des valeurs de différentes natures, sans connaissance préalable de leur type et de leur quantité. Par exemple,

```
std::tuple<int,double,std::string> tup;
```

représente un tuple rassemblant un `int`, un `double` et un objet `std::string`.

Alors que créer un tuple est facile, le manipuler n'est pas toujours évident. A notre disposition, nous avons notamment la fonction `get<I>(t)` qui permet l'extraction du $I^{\text{ème}}$ élément du tuple `t`, et `tuple_element_t<I,T>` qui représente le type du $I^{\text{ème}}$ élément du tuple de type `T`.

Si l'on souhaite appliquer un même traitement à chacun des éléments d'un tuple de taille N , une solution consiste à obtenir la séquence de nombre $0,1,\dots,N-1$ sous la forme d'un pack de nombres (notons le `Is`), et d'appliquer une expansion de ce pack de la manière suivante :

```
f(std::get<Is>(t)...);
```

qui sera développée en :

```
f(std::get<0>(t), std::get<1>(t), ..., std::get<N-1>(t));
```

Pour permettre cette expansion, il faut obtenir le pack d'entiers `Is`. Pour cela, on utilise une fonction générique qui, en prenant un tuple `t` en argument, permet d'obtenir son pack de paramètres `ARGS` par déduction de type. Elle appelle ensuite une autre fonction avec comme arguments le tuple `t` et la séquence de nombres (générée par la fonction `make_index_sequence` dont le paramètre *template* indique la taille de la séquence, obtenue en appliquant l'opérateur `sizeof...` sur `ARGS`) :

```
template <typename... ARGS>
void f(const std::tuple<ARGS...> & t)
{ f_bis(t,std::make_index_sequence<sizeof...(ARGS)>()); } 
```

Par exemple, `f(tup)` induit l'appel `f_bis(tup,std::index_sequence<0,1,2>)`. Il suffit alors d'écrire la fonction `f_bis` avec un pack d'entiers `Is` en paramètre *template* sous la forme suivante :

```
template <typename T, size_t... Is>
void f_bis(const T & t, std::index_sequence<Is...>) { /* Expansion utilisant Is */ }
```

- 4) A partir de cette explication, écrire une surcharge de la fonction chaîne pour les tuples, afin d'appeler la version *variadic* de la fonction chaîne avec comme arguments les éléments du tuple. Tests 4-5
- 5) Supposons maintenant que l'un des éléments d'un tuple soit lui même un tuple, cf. Test 6. D'après notre conception, pas de souci *a priori*, puisque la fonction chaîne est appelée sur chaque élément du tuple, et nous disposons d'une surcharge pour les tuples, donc de manière « récursive », la sérialisation devrait fonctionner. Cependant, le compilateur peut éventuellement appeler la version primaire de chaîne (celle qui est générique et renvoie une exception), ce qui est probablement lié à l'ordre de déclaration des surcharges. Il vous faut donc bien veiller à **déclarer les prototypes en amont des définitions des surcharges** de la fonction chaîne.
- 6) Question subsidiaire : La chaîne produite avec des tuples imbriqués produit à certains moments des espaces multiples (plusieurs caractères espaces à la suite), liés à la fonction chaîne (version *variadic*) où un espace est en trop (soit en début, soit en fin, suivant votre implémentation). Proposer une version qui, au lieu de prendre en paramètre seulement un pack de types, prend d'abord un type, puis un pack de types. De cette manière, vous pourrez écrire la concaténation de tous les éléments sans l'espace superflu.

Exercice 3 - Métaprogrammation

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;
```

Il est possible d'exploiter la capacité de spécialisation des *templates* pour, de manière récursive, obtenir le résultat du calcul. Voici comment procéder avec une fonction générique, illustré avec l'exemple de factorielle $n!$. Partons d'une version dynamique de factorielle.

```
long factorielle(int n) {
    long r = 1;
    for (int i = 2; i <= n; i++) r *= i;
    return r;
}
```

L'utilisation de cette fonction se ferait de la manière suivante : `y = factorielle(5);`

Considérons maintenant une version statique de factorielle, écrite sous la forme d'une fonction générique avec une version primaire (le calcul récursif de factorielle) et d'une spécialisation (qui permet d'arrêter la récursion).

```
template <int N> inline long factorielle() { return (N * factorielle<N-1>()); }
template <> inline long factorielle<0>() { return 1; }
```

L'utilisation de cette fonction se ferait de la manière suivante : `y = factorielle<5>();`

A la place d'une fonction, proposez une structure générique contenant un attribut de classe de type `unsigned long` et qui, de manière similaire, permet d'effectuer le calcul de $n!$ de manière statique. Au final, l'utilisation de cette structure se fera de la manière suivante :

```
unsigned long fact20 = Factorielle<20>::valeur; Test 7
```

2) Calcul de séries de Taylor

L'expression d'une fonction $f(x)$ sous la forme d'une série de Taylor fait intervenir le degré 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 au degré voulu. A 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 au point 1.5 de la série de Taylor de degré 5 de l'exponentielle sera donné par :

```
double val = Exponentielle<5>(1.5);
```

Fournir la série de Taylor des fonctions exponentielle, sinus et cosinus. On rappelle que :

$$\exp(x) \approx \sum_{k=0}^N \frac{x^k}{k!}$$

$$\cos(x) \approx \sum_{k=0}^N (-1)^k \frac{x^{2k}}{(2k)!}$$

$$\sin(x) \approx \sum_{k=0}^N (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

Proposer les structures génériques nécessaires à ces calculs. Test 8-11