

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 ensuite viendront 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 : Utiliser un constructeur *template* prenant en argument la valeur plutôt que la chaîne du type.

- 2) Fournir des surcharges de la fonction chaîne pour les types string, int et double. Pour les deux derniers, utiliser un objet stringstream pour effectuer la conversion. 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 les nombres, on aurait pu utiliser la fonction std::to_string introduite en C++11 qui simplifie la conversion.

Exercice 2 – Variadic template

- 3) Ecrire une surcharge de la fonction `chaine` 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 `chaine` 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> tp;
```

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 *l^{ème}* élément du tuple `t`, et `tuple_element_t<I, T>` qui représente le type du *l^{ème}* élément du tuple de type `T`.

Si l'on souhaite appliquer un même traitement à chacun des éléments d'un tuple, une solution est d'obtenir la séquence de nombre *0, 1, ..., 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 va se développer 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 part d'une fonction qui, à partir d'un tuple, appelle une fonction avec comme arguments le tuple et la séquence de nombres (générée par la fonction `make_index_sequence` dont le paramètre *template* indique la taille de la séquence) :

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

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

```
template <typename T, size_t... Is>
void f_bis(const T & t, std::index_sequence<Is...>)
{ // Expansion de Is pour appeler fonction f }
```

- 4) A partir de cette explication, écrire une surcharge de la fonction `chaine` pour les tuples, afin d'appeler la version *variadic* de la fonction `chaine` avec comme arguments les éléments du tuple. Tests 4-5

Exercice 3 – Initiation à la métaprogrammation

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 `chaine` 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, dans le cas d'un élément tuple dans un tuple, le compilateur appelle la version primaire de `chaine` (celle qui est générique et renvoie une exception).

Dans cette version primaire, au lieu de lever systématiquement une exception, il faut lever celle-ci uniquement si le type `T` n'est pas un tuple. Dans le cas où il s'agit d'un tuple, appeler directement la fonction `chaine_bis` (cf. question 4) pour lever toute ambiguïté et ainsi lancer la conversion.

- 5) Dans un premier temps, définir une variable *template* `is_tuple<T>` qui vaut `false`. Proposer une spécialisation de cette variable *template* pour les tuples, sa valeur sera alors `true`.

A l'aide de cette variable, dans la version primaire de `chaine` (avec l'exception), il est alors possible de faire le test : `if (is_tuple<T>)` et suivant la réponse, lever une exception ou appeler la fonction `chaine_bis` (utiliser `std::tuple_size_v<T>` pour obtenir la taille du tuple). Penser aussi à déclarer la fonction avant son utilisation. Test 6

Attention : Le `if` « classique » a pour conséquence que le compilateur va compiler les deux branches, mais `chaine_bis` n'existe pas pour les types autres qu'un tuple, d'où une erreur de compilation. Pour éviter ce problème, on utilise un `if constexpr` (depuis C++17) qui est évalué à la compilation et ne compile que la branche valide du test.

Question subsidiaire : La chaîne produite avec des tuples imbriqués produit à certain moment des espaces multiples (plusieurs caractères espaces à la suite), liés à la fonction `chaine` 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.