Benjamin Hamon
Sophie Huart
Alexandre Fischer
Sébastien Corbeau
Alain Le
Jeremy Moriaux

OBJET AVANCE

PROGRAMMATION GENERIQUE

CONTEXTE DU PROJET

L'objectif du projet est de créer une bibliothèque logicielle C++ en utilisant les principes de programmation générique et des techniques d'implantations associées vues en cours (traits, méta-programmation, tag dispatching, EDL, etc…).

Notre sujet porte sur les conteneurs multidimensionnels. L'objectif est de trouver des concepts pour gérer et implanter des conteneurs de dimension N quelconque. Pour cela, il faut fournir une implantation d'un tableau à N dimensions et d'en inférer les extensions nécessaires pour gérer de manière homogène d'autres conteneurs en mode N dimensions. Ainsi la librairie finale mettra à disposition de l'utilisateur un certains nombre de fonctions génériques de traitement de tableaux multidimensionnels et un mécanisme d'extension pour intégrer facilement des conteneurs personnalisés.

ORGANISATION

Nous avons créé un dépôt GITHUB : https://github.com/bhamon/genericprogramming

Le dépôt est divisé en plusieurs parties :

- Les répertoires « include » et « src » : ils contiennent les sources de notre librairie. Nous avons défini le namespace « pps::range » pour tous les objets de la librairie.
- Le répertoire « tests » : il contient les différents tests unitaires de la librairie. Chaque fichier cpp se compile séparément et présente les résultats du test associé. Le fichier « tests.h » contient un ensemble de fonctions utilitaires pour réaliser ces tests.
- Les fichiers Makefile : ils permettent de compiler les tests ainsi que la librairie (dans sa version distribuable sous forme d'une librairie statique .lib ou d'une librairie dynamique .so).

DEMARCHE

PREMIERS PAS

Un tableau multidimensionnel au sens large du terme peut prendre diverses formes en C++, comme le montre le code ci-dessous :

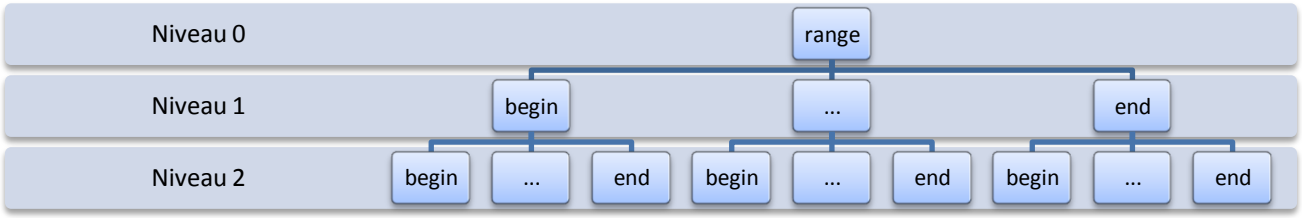
```
std::vector<int> t1;
std::list<std::vector<double**>> t2;
char t3*[20];
```

On constate d'entrée de jeux que l'on peut séparer les conteneurs en trois catégories :

- Les classes de conteneur génériques.
- Les tableaux natifs du langage.
- Les pointeurs sur des séries d'éléments contigus en mémoire.

Dans tous les cas, on peut dégager un modèle en s'inspirant de la « standard library », et plus particulièrement des iterators. Ce modèle sera spécialisé pour prendre en compte les différentes implémentations concrètes.

Ainsi, nous avons fait le choix de considérer une dimension d'un tableau comme un « range », c'est-à-dire une paire d'itérators. Les tableaux multidimensionnels sont donc considérés comme des range en cascade que l'on peut apparenter à un arbre, comme le montre la figure ci-dessous :



La spécification des types de données et des types d'iterators s'effectue par le biais d'un traits dont l'implémentation est localisée dans le fichier « traits.h ». Ce traits a été spécialisé pour les tableaux natifs, les pointeurs sur des éléments contigus en mémoire et les conteneurs de la STL.

La récupération de la paire d'iterators d'une dimension s'effectue par le biais des fonctions « begin » et « end », localisées dans les fichiers « begin.h » et « end.h ». De la même manière que pour le traits, la fonction « begin » a été spécialisée pour les tableaux natifs, les pointeurs et les conteneurs de la STL. Néanmoins, la fonction « end » n'a été spécialisée que pour les conteneurs de la STL. En effet, la récupération du nombre d'éléments présents dans un tableau natif et dans un espace mémoire pointé ne peut être effectuée. La prise en charge des pointeurs et des tableaux natifs devra donc faire l'objet d'un traitement spécial, en passant par exemple par un emballage dans une classe permettant leur gestion (comme nous le verront plus bas).

Grâce à ce panel, il est possible de gérer n'importe quel tableau multidimensionnel formé de tableaux natifs, de pointeurs et de conteneurs de la STL, comme le montre le fichier de test « test_usage.cpp ».

EXTENSIBILITE

La librairie permet de gérer tout type de conteneurs en les apparentant à des range. Pour intégrer un nouveau conteneur au sein de la librairie, il faut procéder en trois étapes simples :

- Spécialisation du traits pour le type utilisateur à gérer.
- Définition des fonctions begin (const et no-const) pour le type en question.
- Définition des fonctions end pour le type en question.

Un exemple d'extension est disponible dans le fichier de test « test_extend.cpp ».

Si un utilisateur de la librairie venait à oublier de renseigner un de ces 3 points d'extension, le code mis en place le préviendra par le biais d'une static_assert liée à un commentaire d'explications lui expliquant son oubli.

AIDE A LA MANIPULATION

En l'état, le parcours descendant dans les différentes dimensions du tableau requiert une écriture un peu lourde. De plus, il est tout aussi difficile de connaitre le type de base manipulé par le tableau multidimensionnel. Pour résoudre ces problèmes, nous avons introduit un second traits disponible dans le fichier « traitsBaseType.h ».

Ce traits apporte donc la possibilité d'inspecter les types consécutifs d'un tableau multidimensionnel en ajoutant la notion de dimension. Un exemple est disponible dans le fichier de test « test_manip.cpp ».

PREMIERE FONCTION : GET

En utilisant tous les concepts précédents, il est donc temps de coder la première fonction de notre librairie qui illustre un cas d'utilisation. Nous avons choisi la fonction get qui permet de retourner un élément à un indice particulier du tableau multidimensionnel. Cette fonction est implémentée dans le fichier « get.h ».

Les prototypes de cette fonction sont les suivants :

```
template<int N, class T, class Indexes>
const traitsBaseType<T, N>::baseType
get(const T& p_range, const Indexes& p_indexes);

template<int N, class T, class Indexes>
traitsBaseType<T, N>::baseType
get(T& p_range, const Indexes& p_indexes);
```

Cette fonction accepte donc en premier paramètre un tableau multidimensionnel, et en second paramètre un tableau unidimensionnel d'indices d'accès à un élément donné. Le code ci-dessous montre un exemple d'utilisation de la fonction :

```
std::vector<std::list<int>> tab;
int e = get(tab, indexes()[2][4]);
```

Dans cet exemple nous utilisons une nouvelle classe nommée « indexes ». Cette classe est un conteneur unidimensionnel permettant de stocker de manière simple les indices d'accès à un élément particulier du tableau multidimensionnel.

NOTION DE VUE

Pour répondre à un problème abordé précédemment, à savoir la prise en charge des pointeurs et des tableaux natifs, nous avons prévu de mettre en place une classe capable d'emballer un tableau multidimensionnel et d'en donner un accès limité. Il s'agit d'un concept de vue qui permet de rendre disponible tout ou partie du tableau contenu. Avec cette technique, nous bénéficions de deux points forts :

- Possibilité de gérer des pointeurs sur des éléments contigus en mémoire, en spécifiant dans la vue la taille du tableau.
- Possibilité d'accéder à des sous-tableaux d'un tableau global, permettant ainsi une manipulation puissante.

Le code ci-dessous montre plusieurs exemples d'utilisation du mécanisme de vue :

```
std::vector<int> myTab;
/*
 On créé une vue sur myTab qui ne rend visible que les éléments
 aux idices 5, 8, 9 et 10.
 Ainsi, la récupération de l'élément [0] retourne l'élément [5].
*/
int value = get(
 view<1>(myTab)[viewSubIndexes()[5][8][9][10]],
 indexes()[0]
);

std::vector<std::list<int>> myTab;
/*
 On créé une vue sur myTab qui ne rend visible que les éléments
 des indices 1 à 21 avec un pas de 2.
 Ainsi, la récupération de l'élément [1] retourne l'élément [3].
*/
int value = get(
 view<2>(myTab)[viewSubRange(1, 21, 2)],
 indexes()[1]
);

// Un tableau 3*3*3.
int*** myTab;
/*
 On créé une vue qui délimite les bornes du tableau pour permettre
 à la fonction "end" d'être appelée au besoin (ce qui ne fonctionne
 pas avec uniquement des pointeurs).
*/
int value = get(
 view<3>(myTab)[viewSubRange(3)][viewSubRange(3)][viewSubRange(3)],
 indexes()[0][0][0]
);
```

Les vues sont partiellement implémentées dans notre librairie. Les différents fichiers en cours de réalisation sont disponible dans les dossiers « include » et « src » et leurs tests dans le dossier « tests ».

POUR ALLER PLUS LOIN

QUELQUES AJOUTS POSSIBLES

Des ajouts possibles pourraient être réalisés. La structure de traits pourrait embarquer des informations supplémentaires, comme des flags qui déterminent si le type en question permet le redimensionnement, la suppression, etc. Ces informations permettraient de bloquer certaines fonctionnalités s'il y a présence d'un conteneur incompatible pour une dimension particulière.

LES PERFORMANCES AVANT TOUT

L'implémentation de la fonction « get » à mis en évidence un point faible de ce système : le manque d'optimisation. En effet, comme nous sommes aussi générique que possible, nous faisons une itération sur les iterators, alors qu'un random acccess sur un std::vector serait beaucoup plus efficace.

Un autre défaut à noter est la récursivité de la fonction qui pèse aussi sur les performances globales.

Enfin, la création des différents objets pour spécifier les indices ou les vues et leur copie au fur et à mesure des appels récursifs est aussi un témoin du manque de rapidité de notre librairie.

Une solution existe pour pallier à tout cela : il s'agit d'écrire une version AST des différentes fonctions de la librairie et de ses opérateurs en conservant l'extensibilité par le biais d'un traits. Nous aurions ainsi à disposition un sous-langage de gestion des tableaux multidimensionnels permettant un gain de lisibilité et de meilleures performances. L'exemple ci-dessous illustre le but visé par la mise en place de cet AST :

```
std::vector<std::list<int>> myTab;

// Récupération d'un élément.
int data = multirange(myTab)[0][4];

// Mise à jour de la valeur d'un élément.
multirange(myTab)[2][2] = 3;

// Récupération d'une portion du tableau.
// Ici on récupère pour la première dimension les indices 1 à 4.
// Puis pour la seconde dimension les indices impairs de 3 à 7.
std::vector<std::list<int>> sub = multirange(myTab)[1:4][3:7:2];

// Mise à zéro d'une partie du tableau.
multirange(myTab)[2:3][5:6] = 0;
```

On pourrait imaginer d'autres opérateurs de manipulation de tableaux multidimensionnels. Un intérêt majeur de cette solution est que chaque opérateur de ce sous-langage pourra bénéficier d'une gestion spécifique à chaque conteneur concret, ce qui permettra à la librairie de gagner en performance.