TP 7 : Expression Templates

Récupérer le code à compléter sur https://www.lri.fr/~bagneres/index.php?section=teaching&page=2015_Et5_DA

On souhaite écrire plusieurs systèmes permettant d'effectuer des sommes, soustractions, multiplications et divisions sur des tableaux (éléments par éléments). Ces opérations ne sont pas disponibles avec `std::vector<T>` mais le sont pour `std::valarray<T>`.

0.1 Pourquoi les opérateurs arithmétiques sont définis pour `std::valarray<T>` et pas pour `std::vector<T>` ?

On suppose que toutes les opérations sur les tableaux se font sur les mêmes types et les mêmes tailles.

1 `vec0<T>` : Une Version Simplifiée De `std::valarray<T>`

Dans le fichier `include/vec0.hpp`, implémenter le code de tous les opérateurs arithmétiques ($+$, $-$, $\times$, $\div$, $-$ unaire). Compiler et exécuter le test `tests/vec.cpp` avec les commandes `cmake ..` et `make && ./test__vec`.

1.1 Si `a` et `b` sont deux `vec0<T>`, combien de parcours de tableaux sont fait pour le calcul de `auto r = (a + b) * -a`. Détailler.

2 `vec1<T>` : Expression Templates !

On souhaite diminuer le nombre de tableaux parcourus.

2.1 Si `a` et `b` sont deux tableaux, combien de parcours de tableaux sont fait au minimum parcourus pour le calcul de `auto r = (a + b) * -a`.

Pour effectuer cette optimisation, on a besoin de connaître toute l'expression. Au lieu d'effectuer le calcul de `a + b` (et donc faire des parcours de tableaux), on va directement retourner un type qui indiquera que le calcul à faire est la somme de `a + b`.

Notre `vec1<T>` a un type `tag` égal à `lazy_expr_tag` (ligne 31). Dans le fichier `include/expression.hpp`, on cherche à surcharger les opérateurs arithmétiques si le type `tag` existe et est égal à `lazy_expr_tag`.

2.2 Dans les opérateurs arithmétiques, expliquer l'utilité, le fonctionnement et les conditions à respecter par le type `T0_is_lazy_expr` (détailler chaque partie).

Dans l'expression `a + b`, l'opérateur `+` doit retourner un type qui stocke une référence (sur `T` constant) pour `a` et `b`. En revanche dans l'expression `(a + b) * c`, il faut que l'opérateur `*` stocke une copie du résultat de l'opérateur `+` et une référence sur `c`.

2.3 Pourquoi ne peut-on pas stocker une référence (sur `T` constant) pour le retour de l'opérateur `+` ?

2.4 Pourquoi ne veut-on pas faire une copie dans tous les cas ?

Pour régler ce souci, les opérateurs détectent si le type à stocker est une *rvalue* (pour faire une copie) ou une *lvalue* (pour créer une `std::reference_wrapper<T const>`. Cela est fait par la fonction `reference_wrapper_if_lvalue`.

2.5 Quel mécanisme de sélection la fonction `reference_wrapper_if_lvalue` utilise-t-elle ?

`lazy_binary_operation` et `lazy_unary_operation` peuvent stocker des `T` et des `std::reference_wrapper<T>`. Dans ces deux cas, les accesseurs sur les données stockées retournent des `T` en forçant le type de retour à `remove_reference_wrapper<T>::type`.

2.6 Quel mécanisme la structure `remove_reference_wrapper` utilise-t-elle pour enlever le `std::reference_wrapper` lorsque cela est nécessaire ?

La bibliothèque standard propose `std::pair<T0, T1>` et la fonction `std::make_pair(T0, T1)`. On a la même chose ici avec `lazy_plus<T0, T1>` et `make_lazy_plus(T0, T1)`.

2.7 Quel est l'intérêt de passer par une fonction pour créer un objet plutôt que d'appeler directement son constructeur ?

Dans `include/expression.hpp`, implémenter les fonctions `display_expr` qui permettent d'afficher l'expression.

Dans `include/vec1.hpp`, on souhaite retourner la taille du tableau qui est le résultat du calcul de l'expression évaluée. Rappel : on suppose que tous les tableaux de l'expression font la même taille.
Implémenter les fonctions `get_size`.

Dans `include/vec1.hpp`, on récupérer le types des éléments du tableau résultat. Rappel : on suppose que tous les éléments des tableaux de l'expression sont de même type.
Implémenter les structures `get_element_type`.

Dans `include/vec1.hpp`, évaluer l'expression (en effectuant le calcul). Implémenter les fonctions `compute_from_expr`.

2.8 Pour l'expression `(a + b) * -a`, quel est le code équivalent (avec des boucles) si les tableaux sont des `vec0<T>` ? si les tableaux sont des `vec1<T>` ?

3 `vec2<T>` : Avec Un Type Dynamique ?

`vec2<T>` utilise le même principe que `vec1<T>` mais en remplaçant le typage statique par un typage dynamique. Les différentes opérations arithmétiques sont stockées dans des `operation_t`. Le type de l'arbre des opérations est `ast_t<T>` (*abstract syntax tree*). Il s'agit d'un "*typedef template*" pour `tree<operation_t<T>>`.

3.1 Quels sont les intérêts d'avoir le typage de l'expression dynamique ?

3.2 Quels sont les différents constructeurs `public` (constructeurs nommés inclus) de la classe `operation_t` ? Quel est l'intérêt d'utiliser les constructeurs nommés ici ?

3.3 Pourquoi l'héritage n'est pas utilisée ici pour les différentes opérations ?

On définira que les opérateurs arithmétiques uniquement pour les `operation_t` (et pas pour les `vec2<T>`). L'opérateur `+` unaire tel que définit permet de convertir un `vec2<T>` en `operation_t`.

3.4 Quelle est la différence entre `ast_t<vec2<T>> operator +(vec2<T> && a)` et `ast_t<vec2<T>> operator +(vec2<T> const & a)` ?

3.5 Expliquer le rôle des données membres `p_data` et `m_data` de la classe `operation_t`. Donner des cas d'utilisation.

3.6 Expliquer l'implémentation de `ast_t<T> operator +(ast_t<T> const & a, ast_t<T> const & b)`. Détailler les types créés et les conversions.

Dans `include/vec2.hpp`, implémenter la fonction `display_expr`.
Dans `include/vec2.hpp`, implémenter les fonctions `get_size`.
Dans `include/vec2.hpp`, implémenter la fonction `compute_from_expr`.

4 Au Sujet Des Performances

4.1 Expliquer les performances obtenues en distinguant le temps de calcul de l'expression et celui du vrai calcul.

4.2 Les performances de `vec2<T>` sont mauvaises, que faire pour les améliorer significativement.