

TD8 : Patrons

On désire développer un ensemble de classes qui permettent de stocker des objets de n'importe quel type (tableaux d'objets, liste chaînée d'objets, pile d'objets, *etc*). Les objets peuvent par exemple être des entiers, des réels, des fractions, des événements, *etc*. On désigne les classes qui permettent de contenir d'autres objets par le terme générique de « **conteneur** ».

On souhaite que chaque conteneur implémenté ait un mode d'utilisation commun et donc une **interface commune obligatoire** à tous les conteneurs.

On appellera « **taille d'un conteneur** » le nombre d'objets qu'il contient. Un conteneur est **vide** lorsqu'il ne contient pas d'objets. Le **premier objet d'un conteneur sera le 0^{ième} objet** du conteneur (comme dans un tableau).

Partie 1

Dans la suite, on appelle T, le type des objets contenus dans les conteneurs. L'**interface commune** à chaque conteneur est la suivante :

- `unsigned int size() const;` qui renvoie la taille du conteneur.
- `bool empty() const;` qui renvoie vrai si le conteneur est vide et faux sinon.
- `T& element(unsigned int i);` qui renvoie le i^{ème} élément d'un conteneur.
- `const T& element(unsigned int i) const;` qui renvoie le i^{ème} élément d'un conteneur.
- `T& front();` qui renvoie une référence sur le premier objet contenu dans le conteneur.
- `const T& front() const;` qui renvoie une référence sur le premier objet contenu dans le conteneur.
- `T& back();` qui renvoie une référence sur le dernier objet contenu dans le conteneur.
- `const T& back() const;` qui renvoie une référence sur le dernier objet contenu dans le conteneur.
- `void push_back(const T& x);` qui ajoute un objet au conteneur après le dernier objet.
- `void pop_back();` qui retire le dernier objet du conteneur.
- `void clear();` qui retire tous les objets du conteneur.

Question 1 (30 min) : Lire l'énoncé en entier. Utiliser UML pour modéliser l'ensemble des classes du problème.

Dans la suite, vous déclarerez et définirez chaque constructeur, chaque destructeur, chaque membre donnée et chaque fonction membre (de l'interface obligatoire) partout où cela est nécessaire. Vous définirez aussi un constructeur de recopie et un opérateur d'affectation partout où cela vous semble nécessaire.

Question 2 (30 min) : Implémenter la classe abstraite « **contener** » modèle de tous les autres conteneurs. Exploiter au mieux le design pattern "template method" pour utiliser le moins de méthodes virtuelles pures possible.

Question 3 (40 min) : Implémenter une classe « **vector** » qui sera basée sur le modèle « **conteneur** ». Cette classe utilisera en attribut un tableau de T alloué dynamiquement pour composer ses éléments. Pour cela, On supposera que le type T dispose d'un constructeur sans argument.

Le constructeur « principal » prendra en argument la taille initiale du tableau et la valeur avec laquelle les objets initialement présents dans le tableau doivent être initialisés. On étudiera éventuellement la possibilité d'avoir des arguments par défaut.

Surcharger en plus l'opérateur « `operator[]` » qui permettra de modifier ou de lire la valeur d'un élément particulier du tableau. Justifier les arguments et le type de retour de cet opérateur.

Question 4 (20 min) : En utilisant astucieusement le design pattern "adapter", implémenter une classe « `stack` » qui sera aussi un type de conteneur mais qui ne devra avoir comme **seule interface possible** :

- `bool empty() const;`
- `void push(const T& x);` qui empile un objet dans la pile.
- `void pop();` qui dépile le dernier élément empilé de la pile.
- `unsigned int size() const;`
- `T& top();` qui renvoie une référence sur le dernier objet empilé de la pile
- `const T& top() const;`
- `void clear();`

On réfléchira éventuellement à la possibilité de pouvoir adapter n'importe quel conteneur pour implémenter cette classe.

Partie 2

Question 6 (50 min): Implémenter le design pattern "itérateur" pour les classes `vector` et `stack` :

- Pour accéder à l'élément pointé en cours, on utilisera l'opérateur `operator*`.
- Pour se déplacer pour pointer vers l'élément suivant, on utilisera l'opérateur `operator++`.
- Afin de comparer deux itérateurs, on surchargera les opérateurs `operator==` et `operator!=` : on suppose que deux itérateurs sont égaux si ils désignent le même élément.
- Implémenter une fonction `swap` qui permet d'inverser le contenu des emplacements désignés par deux itérateurs.
- Pour la classe `tableau`, on implémentera la fonction `begin()` qui renvoie un itérateur désignant le premier élément.
- Pour la classe `tableau`, on implémentera aussi la fonction `end()` qui renvoie un itérateur désignant **l'élément suivant le dernier élément**.
- Avec un simple copier/coller et quelques modifications, on implémentera aussi un type "`const_iterateur`" ainsi que les méthodes `begin()` et `end()` correspondantes.

Question 7 (20 min) : Implémenter la fonction « `minimum_element` » qui prend en arguments deux itérateurs `it1` et `it2` de n'importe quel conteneur et qui permet de renvoyer un itérateur désignant l'élément minimum dans le conteneur entre les itérateurs `it1` et `it2`, `it2` non compris par rapport à l'opérateur `operator<`. On supposera pour cela que cet opérateur a été surchargé pour le type d'élément contenu dans le conteneur.

Question 8 (35 min) : Implémenter la fonction « `minimum_element` » qui prend en arguments deux itérateurs `it1` et `it2` de n'importe quel conteneur, ainsi qu'un prédicat binaire définissant un ordre sur les éléments (design pattern "Strategy"). La fonction permet de renvoyer un itérateur désignant l'élément minimum par rapport au prédicat binaire dans le conteneur entre les itérateurs `it1` et `it2`, `it2` non compris.

Le prédicat binaire doit renvoyer **true** ou **false**. Il pourra être soit une fonction prenant en arguments deux objets du type de ceux contenus dans le conteneur, soit un objet fonction dont l'opérateur `operator()` prend en arguments deux objets du type de ceux contenus dans le conteneur.

Pour aller plus loin...

(ces exercices ne sont pas à traiter durant la séance de TD)

Utilisation de la classe standard allocator

On suppose maintenant que le type `T` ne dispose pas forcément d'un constructeur sans argument. Modifier votre classe `vector` de manière à prendre cet aspect en compte.

Pour cela, on utilisera la classe standard allocator du C++ (voir poly) qui permet de séparer l'allocation/désallocation d'une zone mémoire de la construction/destruction d'objets (ou de tableaux d'objets) dynamique.

Le constructeur « principal » prendra en argument la taille initiale du tableau et la valeur avec laquelle les objets initialement présents dans le tableau doivent être initialisés. On étudiera éventuellement la possibilité d'avoir des arguments par défaut.

On appellera « **capacité d'un tableau** » le nombre maximum d'objets qu'il peut contenir avant de devoir refaire une réallocation. Initialement, cette capacité est égale à la taille du tableau. Cependant, on pourra changer la capacité (sans pour autant changer la taille) grâce à la méthode `void reserve(unsigned int n)`. Cette méthode n'a une action que si `n` est strictement supérieur à la taille du tableau (dans le cas contraire, la méthode ne fait rien). Le tableau dispose alors d'une réserve supplémentaire qu'il peut utiliser lorsque le nombre d'éléments augmente (par ex avec la méthode `push_back`) sans devoir pour autant faire une réallocation. Implémenter la méthode `unsigned int capacity() const` permettant de connaître la capacité du vecteur.

On ré-implémentera aussi la méthode `void resize(unsigned int t=0, const T& initialize_with=T());` qui permet de changer le nombre d'éléments du tableau. Si la taille diminue, la capacité ne change pas : les objets qui doivent disparaître du tableau sont "détruits" mais les cellules qui ne sont plus utilisées sont gardées en réserve (elles ne sont pas désallouées).

Si la taille augmente, les nouvelles cases utilisées sont initialisées avec la valeur `initialize_with`. Si la nouvelle taille est inférieure à la capacité il n'y a pas de réallocation. Si la nouvelle taille est supérieure à la capacité, un nouveau tableau est réalloué et la capacité devient égale à la nouvelle taille.

De même, la méthode ne provoque pas de réallocation tant que la capacité le permet (`push_back`). La méthode `clear` ne provoque pas non plus une désallocation mais seulement une destruction des objets du tableau.

La classe list

Implémenter une classe « `list` » qui sera une liste doublement chaînée.

Le constructeur « principal » prendra en argument la taille initiale de la liste et la valeur avec laquelle les objets initialement présents dans la liste doivent être initialisés. On étudiera éventuellement la possibilité d'avoir des arguments par défaut.

Définir les fonctions `void push_front(const T& x)` et `void pop_front(const T& x)` qui ajoute ou retire un élément en tête de liste.

Définir la fonction `bool remove(const T& x)` qui retire le premier élément de la liste qui a la valeur `x`. La fonction renverra `true` si l'opération réussit et `false` sinon (si `x` n'existe pas dans la liste).