TD 2 : Conteneurs, Itérateurs

Objectifs pédagogiques :

- **introduction à la lib standard**
- **gestion mémoire, allocations**
- **itérateurs du C++, range**
- **vecteur<T>, liste<T>, map<K,V>**

Introduction

Dans ce TD, nous allons réaliser en C++ des classes `Vector<T>`,`List<T>`,`Map<K,V>` offrant le confort habituel de ces conteneurs classiques. Cet exercice est à vocation pédagogique, on préférera en pratique utiliser les classes standard, qu'on trouvera dans les header `<vector>`, `<list>`, `<unordered_map>`.

L'objectif de l'exercice est de bien comprendre les conteneurs standards du c++ et leur API.

Pour être sûr de ne pas entrer en conflit avec d'autres applications, nous utiliserons le namespace `pr` pour notre implémentation.

1.1 Vecteur : stockage contigü.

Un vecteur stocke de façon contigüe en mémoire des données. C'est une des structures de données les plus simples, mais pour cette raison ça reste un bon choix pour de nombreuses applications.

Question 1. Ecrivez une classe `Vector<T>` en respectant les contraintes suivantes :

- Le vecteur est muni d'un pointeur vers l'espace mémoire alloué pour stocker les données
- Le vecteur est muni d'une taille `size`, qui indique le remplissage actuel
- Le vecteur a une taille d'allocation, toujours supérieure ou égale à size
- Les objets de type T ajoutés sont copiés dans l'espace mémoire géré par le vecteur

On fournira pour cette classe les operateurs et fonctions suivantes :

- `Vector(int size=10)` : un constructeur qui prend la taille d'allocation initiale
- `Ṽector()` un destructeur
- `operator[]` dans ses deux variantes (const ou non), pour consulter ou modifier le contenu.
- `size_t size() const` la taille actuelle (nombre d'éléments)
- `bool empty() const` vrai si la taille actuelle est 0
- `void push_back (const T& val)` : ajoute à la fin du vecteur, peut nécessiter réallocation et copie.

1.2 Liste : stockage par Chainon.

Une liste simplement chaînée stocke les données dans des chaînons.

Question 2. Ecrivez une classe `Liste<T>`.

- Une Liste est munie d'un pointeur vers le premier chaînon qui la constitue (ou `nullptr` si elle est vide)
- La liste ne stocke pas sa taille, il faut la recalculer
- Un Chainon est composé d'un attribut de type T (le data) et d'un pointeur vers le prochain Chainon (ou nullptr).

On fournira pour ce cette classe les operateurs et fonctions suivantes :

- `List()` : un constructeur par défaut pour la liste vide

- `~List()` un destructeur, qui libère tous les chaînons
- `size_t size() const` la taille actuelle (nombre d'éléments)
- `bool empty() const` vrai si la liste est vide
- `void push_back (const T& val)` : ajoute à la fin de la liste
- `void push_front (const T& val)` : ajoute en tête de la liste

1.3 Itérateurs.

Question 3. Quelles opérations doit fournir un itérateur du C++ ? Comment itérer de façon standard sur un conteneur ?

Question 4. Ajoutez ces mécanismes à votre Liste. En quoi consiste un itérateur ?

Question 5. Ajoutez ces mécanismes à votre Vector. En quoi consiste un itérateur ?

Question 6. Expliquez la différence entre les `iterator` et `const_iterator`. Expliquez comment il faut modifier Vector et List pour permettre des itérations const.

Question 7. Ecrivez une fonction qui affiche le contenu d'un Vector. Comment la modifier à l'aide de template pour permettre d'afficher également une Liste ? Utilisez la fonction dans un main.

TME 2 : Conteneurs, Itérateurs, Lib Standard

Objectifs pédagogiques :

- **conteneurs**
- **itérateurs**
- **map**

1.1 Liste, Vecteur

Question 1. En suivant les indications du TD 2, implanter les classes génériques Liste<T> et Vector<T> et leurs itérateurs. Elles doivent pouvoir s'utiliser avec ce main (générique) :

main_template.cpp

```
#include "List.h"
#include "Vector.h"
#include <iostream>

using namespace std;
using namespace pr;


template< template<typename> class Container, typename T>
void show_const (const Container<T> & vec) {
      std::cout << "const:" << std::endl;
      for (const auto & val : vec) {
            std::cout << val;
      }
      std::cout << std::endl;
}


template< template<typename> class Container, typename T>
void show (Container<T> & vec) {
      std::cout << "avant:" << std::endl;
      for (auto & val : vec) {
            std::cout << val << " ";
            ++val;
      }
      std::cout << "apres:" << std::endl;
      show_const(vec);
      std::cout << std::endl;
}



template< template<typename> class Container, typename T>
int test (Container<T> & vec) {

      for (int i=0; i < 15 ; i++) {
            vec.push_back(i);
      }

      std::cout << vec << "size " << vec.size() <<std::endl;

      show_const(vec);
      show(vec);

      return 0;
}
```

```
int main_t () {
      Vector<int> vec;
      test(vec);
      std::cout << vec[4] << std::endl;
      vec[4]++;
      std::cout << vec[4] << std::endl;

      List<int> list;
      test(list);
      return 0;
}
```

1.2 std::vector, std::pair

On part du programme suivant, qui essaie de compter combien de mots différents sont utilisés dans un livre.

Compte le nombre mots différents

```
#include <iostream>
#include <fstream>
#include <regex>
#include <algorithm>
#include <vector>
#include <chrono>

using namespace std;

int main_p () {
       vector<string> words;
       words.reserve(5000);

       ifstream input = ifstream("/tmp/WarAndPeace.txt");

      std::chrono::steady_clock::time_point start = std::chrono::steady_clock::now();
      std::cout << "Parsing War and Peace" << endl;

      std::string s;
      // une regex qui reconnait les caractères anormaux (négation des lettres)
      regex re( R"([^a-zA-Z])");
      while (input >> s) {
            // élimine la ponctuation et les caractères spéciaux
            s = regex_replace ( s, re, "");
            // passe en lowercase
            std::transform(s.begin(),s.end(),s.begin(),::tolower);

            // cherchons si le mot est déjà  présent
            auto it = words.begin();
            while (it != words.end()) {
                  if (*it == s) {
                        // trouvé
                        break;
                  }
                  ++it;
            }
            if (it != words.end()) {
                  // déjà  trouvé
                  continue;
```

```
            } else {
                  words.push_back(s);
            }
      }
      input.close();

      std::cout << "Finished Parsing War and Peace" << endl;

      std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
   std::cout << "Parsing with took "
         << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
         << "ms.\n";

   std::cout << "Found a total of " << words.size() << " words." << std::endl;

   return 0;
}
```

Question 2. Exécutez le programme sur le fichier WarAndPeace.txt fourni. Combien y a-t-il de mots différents ?

Question 3. Modifiez le programme pour qu'il calcule le nombre d'occurrences de chaque mot. Pour cela, on adaptera le code pour utiliser un vecteur qui stocke des `pair<string,int>` au lieu de stocker juste des string. Afficher le nombre d'occurrences des mots "war", "peace" et "toto".

Question 4. Que penser de la complexité algorithmique de ce programme ? Quelles autres structures de données de la lib standard aurait-on pu utiliser ?

1.3 Table de Hash

On souhaite à présent implanter une table de hash assez simple, en appui sur les classes Liste<T> et Vector<T> qu'on a développé plus haut.

On rappelle les principes d'une table de hash :

- La table stocke un vecteur de taille relativement grande appelé *buckets*.
- Dans chaque case du vecteur, on trouve une liste de paires (clé,valeur)
- Pour rechercher une entrée à partir d'une clé k, on hash la clé, ce qui rend un entier $hash(k)$ dont la valeur dépend du contenu de la clé.
- On cherche dans le bucket d'indice `hash(k) % buckets.size()`, un élément de la liste dont la clé serait égale (au sens de ==) avec la clé k recherchée.
- Si on le trouve, on peut exhiber la valeur qui lui est associée, sinon c'est qu'il n'est pas présent dans la table.

Question 5. Ecrire une classe générique `HashTable<K,V>` où K est un paramètre qui donne le type de la clé, et V donne le type des valeurs.

On pourra dans l'ordre :

- définir le cadre de la classe, ses paramètres génériques
- définir un type `Entry` pour contenir les paires clés valeur ; les clés sont stockées de façon const, pas les valeurs
- ajouter un attribut typé `Vector< List< Entry > >` dans la classe
- définir un constructeur prenant une taille, qui initialise le vecteur avec des listes vides dans chaque bucket

Ensuite, définir un itérateur (version non const), son opération de déréférencement, sa comparaison, son incrément.

- il porte une réference vers la table *buckets*

- il porte un indice ou un itérateur *vit* dans cette table, désignant la liste (*bucket*) qu'il explore actuellement
- il porte un itérateur *lit* de liste, qui pointe l'élément courant dans la liste que désigne l'itérateur
- pour l'incrémenter, on incrémente d'abord *lit*, si l'on est au bout de la liste, on se décale sur *vit* à la recherche d'une case non vide. *lit* devient alors la tête de cette liste.
- pour la comparaison d'égalité, on peut se contenter de comparer les itérateurs *lit*
- le déréférencement rend une *Entry*, celle qui est pointée par *lit*

Enfin définir les méthodes utilisant ces itérateurs :

- Définir les opérations `begin()` et `end()` de la table de hash.
- Définir `iterator find (const K & key)` qui rend un itérateur correctement positionné, ou *end*() si la clé n'est pas trouvée. Pour calculer la valeur de hash de l'objet *key*, on utilisera `size_t h = std::hash<K>()(key);`. Cette fonction ne doit pas itérer toute la table, seulement le "bucket" approprié.
- Définir `std::pair<iterator,bool> insert (const Entry & entry)` qui essaie d'insérer la paire clé valeur fournie dans la table.
  - Si la clé était déjà présente dans la table, le booléen rendu est faux, et l'itérateur pointe l'entrée qui existait déjà dans la table (qui n'a pas été modifiée).
  - Si la clé n'était pas encore présente, le booléen rendu est vrai, et l'itérateur pointe l'entrée qui vient d'être ajoutée.

1.4 Mots les plus fréquents

Question 6. Utiliser une table de hash associant des entiers (le nombre d'occurrence) aux mots, et reprendre les question où l'on calculait de nombre d'occurrences de mots avec cette nouvelle structure de donnée.

Question 7. Après avoir chargé le livre, initialiser un `std::vector<pair<string,int> >`, par copie des entrées dans la table de hash. On pourra utiliser le contructeur par copie d'une range : `vector (InputIterator first, InputIterator last)`.

Question 8. Ensuite trier ce vecteur par nombre d'occurrences décroissantes à l'aide de `std::sort`. On lui passe les itérateurs de début et fin de la zone à trier, et un prédicat binaire. Voir l'exemple suivant.

exemple sort et lambda

```cpp
#include <vector>
#include <string>
#include <algorithm>

class Etu {
      public :
      std::string nom;
      int note;
};

int main_sort () {
      std::vector<Etu> etus ;
      // plein de push_back de nouveaux étudiants dans le désordre

      // par ordre alphabétique de noms croissants
      std::sort(etus.begin(), etus.end(), [] (const Etu & a, const Etu & b) { return a.nom <
           b.nom ;});
      // par notes décroissantes
      std::sort(etus.begin(), etus.end(), [] (const Etu & a, const Etu & b) { return a.note
          > b.note ;});
      return 0;
```

```
}
```

Question 9. Si la table de hash est trop pleine, on aura beaucoup de collisions de hash, c'est à dire que les listes stockées dans chaque bucket deviennent de plus en plus longues. Ecrivez une fonction `grow` dans votre table de hash, qui double la taille du vecteur de buckets, et réinsère les éléments dans le résultat. Quelle est la complexité de cette réindexation ?