TD 2 : Mémoire, Conteneurs, Map

Objectifs pédagogiques :

- **gestion mémoire, allocations**
- **vecteur<T>, liste<T>, map<K,V>**
- **introduction à la lib standard**

1.1 Copie et Affectation

On considère, une classe String définie comme suit :

String.h

```
#pragma once
#include <cstddef> // size_t
#include "strutil.h"

namespace pr {

class String {
      const char * str;
public:
      // ctor : copie
      String(const char *cstr=""): str(newcopy(cstr)){};
      // dtor : libère
      virtual ~String() { delete [] str;}

      // taille
      size_t length() const { return pr::length(str);};
};

} // fin namespace pr
```

Question 1. Expliquez les problèmes que cette version pose sur cet exemple.

```
int main() {
      String abc = "abc";
      {
            String bcd(abc);
      }

      std::cout << abc << std::endl;

      String def = "def";
      def = abc;

      std::cout << abc << " et " << def << std::endl;
}
```

Question 2. Modifiez la classe String, afin qu'elle se comporte bien : ajoutez les opérateurs d'affectation et la construction par copie.

Conteneurs

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.

1.2 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 3. 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
- `~Vector()` un destructeur
- `T& operator[](size_t index)` et `const T& operator[](size_t index) const` 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.

1.3 Liste : stockage par Chainon.

Une liste simplement chaînée stocke les données dans des chaînons.

Question 4. Ecrivez une classe `List<T>`.

- Un Chainon est composé d'un attribut de type T (le data) et d'un pointeur vers le prochain Chainon (ou nullptr).
- 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

On fournira pour cette classe les operateurs et fonctions suivantes :

- `List()` : un constructeur par défaut pour la liste vide
- `~List()` un destructeur, qui libère tous les chaînons
- `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
- `size_t size() const` la taille actuelle (nombre d'éléments)
- `bool empty() const` vrai si la liste est vide
- `T& operator[](size_t index)` dans sa variante permettant de modifier le contenu.

1.4 Table de Hash

On souhaite à présent implanter une table de hash assez simple, en appui sur les classes `forward_list<T>` et `vector<T>` du standard, et qui sont des versions plus complètes des classes qu'on vient de développer.

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< forward_list< Entry > >` dans la classe
- définir un constructeur prenant une taille, qui initialise le vecteur avec des listes vides dans chaque bucket

Ensuite définir les méthodes d'accès suivantes, dont la signature est proche de l'API proposée en Java (l'API C++ de `std::unordered_map` s'appuie sur les itérateurs, que l'on présentera au TD3).

- Définir `V* get(const K & key)` qui rend l'adresse de la valeur associée à la clé si on la trouve, ou `nullptr` dans le cas contraire. 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 `bool put (const K & key, const V & value)` qui ajoute l'association $(key, value)$ à la table. La fonction rend vrai si la valeur associée à key a été mise à jour dans la table, et faux si on a réalisé une insertion (la clé n'était pas encore dans la table).
- Une fonction `size_t size() const` qui rend la taille actuelle.

Question 6. Si la taille actuelle est supérieure ou égale à 80% du nombre de buckets, la table est considérée surchargée : la plupart des accès vont nécessiter d'itérer des listes. On souhaite dans ce cas doubler la taille d'allocation (nombre de buckets). Ecrivez une fonction membre `void grow()` qui agrandit une table contenant déjà des éléments. Quelle est la complexité de cette réindexation ?

TME 2 : Conteneurs, Map, Lib Standard

Objectifs pédagogiques :

- **conteneurs**
- **map**
- **algorithm, lib std**

1.1 std::vector, std::pair

On part du programme suivant, qui extrait et compte les mots contenu dans un livre.

Compte le nombre mots

```cpp
#include <iostream>
#include <fstream>
#include <regex>
#include <chrono>

int main () {
      using namespace std;
      using namespace std::chrono;

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

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

      size_t nombre_lu = 0;
      // prochain mot lu
      string word;
      // une regex qui reconnait les caractères anormaux (négation des lettres)
      regex re( R"([^a-zA-Z])");
      while (input >> word) {
            // élimine la ponctuation et les caractères spéciaux
            word = regex_replace ( word, re, "");
            // passe en lowercase
            transform(word.begin(),word.end(),word.begin(),::tolower);

            // word est maintenant "tout propre"
            if (nombre_lu % 100 == 0)
                  // on affiche un mot "propre" sur 100
                  cout << nombre_lu << ": "<< word << endl;
            nombre_lu++;
      }
      input.close();

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

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

   cout << "Found a total of " << nombre_lu << " words." << endl;

   return 0;
}
```

Question 1. Exécutez le programme sur le fichier WarAndPeace.txt fourni. Combien y a-t-il de mots ?

Question 2. Modifiez le programme pour compter le nombre de mots différents que contient le texte. Pour cela on propose dans un premier temps de stocker tous les mots rencontrés dans un vecteur, et de traverser ce vecteur à chaque nouveau mot rencontré pour vérifier s'il est nouveau ou pas. 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 précédent 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 ?

1.2 Table de Hash

Question 5. En suivant les indications du TD 2, implanter la classe générique HashMap<K,V>.

1.3 Mots les plus fréquents

Question 6. Utiliser une table de hash `HashMap<string,int>` associant des entiers (le nombre d'occurrence) aux mots, et reprendre les questions 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` et afficher les dix mots les plus fréquents.

`std::sort` prend les itérateurs de début et fin de la zone à trier, et un prédicat binaire. Voir l'exemple suivant.

exemple sort et lambda

```
#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. S'il vous reste du temps, remplacer les `forward_list` et `vector` utilisés par votre classe HashMap par vos propres implantations `List,Vector` comme discuté en TD. Ajouter les opérations nécessaires au fonctionnement de votre table de hash dans ces classes.