TD 3 : Iterator, Algorithm

Objectifs pédagogiques :

- **for each** $c++$
- **iterator, const_iterator**
- **algorithm**

Introduction

Dans ce TD, nous allons ajouter des itérateurs à nos conteneurs et explorer quelques algorithmes de la lib standard.

1.1 Boucle For Each

1.2 Itérateurs.

On rappelle l'expansion syntaxique d'une structure de contrôle "for each" ou "range for" en C++11.

```
// boucle "foreach"
for (DeclType i : cont) {
      // body
}

// Expansion C++11
{
  for(auto it = cont.begin(),_end = cont.end(); it != _end; ++it)
  {
      DeclType i = *it;
            // body
  }
}
```

Dans ce code, `cont` est un conteneur, et `DeclType` est une déclaration de type compatible avec le contenu (`T, T&, const T&` si `cont` est un `conteneur<T>`).

Question 1. En déduire les méthodes que doit fournir un conteneur du C++, et les contraintes sur le type de retour de ces méthodes. Quelles opérations doit fournir un itérateur comme "it" dans ce code ?

Question 2. On considère le $Vector < T >$ développé au précédent TD. Si son itérateur se réduit simplement à un pointeur sur le contenu, montrez que la sémantique des opérateurs $!=, *, ++$ se conforme au contrat attendu. Enrichissez votre classe pour permettre une boucle "range-for" dessus.

Question 3. On considère la liste simplement chaînée `List<T>` développée au précédent TD. Si l'on suppose que son itérateur se limite à stocker à un pointeur de `Chainon`, comment aménager les opérateurs $!=, *, ++$ dans cette classe ? Enrichissez votre classe `List<T>` pour permettre une boucle "range-for" dessus.

1.3 Itérations constantes

Question 4. On considère la boucle suivante, qui permet d'afficher un vecteur. Le compilateur se plaint de problèmes liés au fait que le vecteur est *const*, et refuse de compiler ce code. Expliquez le problème qui se pose ici et ce qu'il faut faire pour le corriger.

```
template<typename T>
ostream & operator<< (ostream & os, const Vector<T> & vec) {
   for (const T & elt : vec) {
```

```
      os << elt << ", ";
   }
   os << endl;
}
```

Question 5. Expliquez les différences entre les signatures des opérations d'un `iterator` et d'un `const_iterator`. Expliquez comment il faut modifier Vector et List pour permettre des itérations const.

1.4 Algorithme et itérateur

Question 6. Ecrivez une fonction générique

```
template<typename iterator, typename T>
iterator find (iterator begin, iterator end, const T& target)
```

qui rend un itérateur *it* satisfaisant `target==*it` si l'on en trouve entre `begin` et `end`, ou `end` dans le cas contraire. Comment l'invoquer sur une `List<int>` ? Sur un `Vector<string>` ?

Question 7. La fonction `find_if` du standard prend un prédicat *pred* en troisième paramètre, c'est à dire une fonction, un foncteur, ou une lambda qui offre la signature : `bool matches (const T & elt)`.

Elle rend comme la fonction `find` un itérateur *it* satisfaisant `pred(*it)` si l'on en trouve entre `begin` et `end`, ou `end` dans le cas contraire.

Comment l'invoquer sur un `Vector<string>` pour trouver une string de trois caractères de long ?

Question 8. Comment l'invoquer pour trouver une string de longueur n arbitraire, la valeur de n étant connue au runtime ?

TME 3 : Iterator, Algorithm

Objectifs pédagogiques :

- **iterator**
- **algorithm**
- **unordered_map**

On poursuit dans ce TME le travail démarré en séance 2. Inutile donc de créer un nouveau projet.

1.1 Algorithme

Question 1. Ecrivez une fonction générique `size_t count (iterator begin, iterator end)` qui compte le nombre d'éléments entre *begin* et *end*.

Question 2. Ecrivez une fonction générique `size_t count_if_equal (iterator begin, iterator end, const T & val)` qui compte le nombre d'éléments entre *begin* et *end* qui sont égaux à *val* au sens de ==.

Question 3. Invoquez ces fonctions dans votre programme qui compte les mots, et contrôlez le résultat : par exemple une invocation à *count* et la valeur rendue par *size* doivent être cohérentes.

1.2 Itérateurs sur HashMap

Question 4. Enrichir la classe `HashMap<K,V>` développée au précédent TD, pour permettre d'itérer sur son contenu. Le type contenu (i.e. ce qui est rendu par $*it$ sur un itérateur) sera une entrée dans la table, i.e. une paire clé valeur.

On donne les indications suivantes pour définir un itérateur (version non const), son opération de déréférencement, sa comparaison, son incrément.

En attribut,

- 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

En méthodes,

- `iterator & operator++()` 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.
- `bool operator!=(const iterator &other)` pour la comparaison d'inégalité, on peut se contenter de comparer les itérateurs *lit*
- `Entry & operator*()` le déréférencement rend une *Entry*, celle qui est pointée par *lit*

Question 5. A l'aide de cet itérateur, recopier les entrées de votre table de hash dans un `std::vector` contenant des paires (clé,valeur).

Question 6. Enfin trier ce vecteur, par fréquence décroissante, et afficher les dix mots les plus fréquents du livre.

1.3 unordered_map

La classe `std::unordered_map<K,V>` est une table de hash plus complète que celle que l'on a implémenté. Son API utilise des itérateurs pour retrouver une clé avec *find* (joue le rôle de *get* dans notre implantation) et insérer une nouvelle paire clé valeur avec *insert* (joue le rôle de *put* de notre implantation).

- `iterator find (const K & key)` rend un itérateur correctement positionné, ou *end*() si la clé n'est pas trouvée.
- `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.

Question 7. Dans une copie de votre main, substituer une `unordered_map` à votre map actuelle et corriger les problèmes liés à cette API différente de la notre.

Question 8. On souhaite à présent inverser notre table associative : on souhaite créer une `unordered_map<int,forward_list<string> >`, où pour une fréquence donnée, on trouve la liste des mots ayant cette fréquence dans le livre. Construisez cette table, puis afficher tous les mots qui ont N occurrences (choisissez des valeurs de N).

Pour construire la table, on itère la table actuelle les paires (k, v), et l'on ajoute dans la table cible à la liste désignée par la clé v avec $push\_front(k)$.

Question 9. Sur ce même modèle, si l'on a une classe `Personne` avec différents champs `nom, prenom, age, sexe, ...` et un `vector<Personne>`. Expliquez comment construire efficacement les sous-groupes de Personnes qui ont le même age, ou le même prénom.

NB: Cette opération orientée-objet très utile en pratique qui construit des classes d'équivalence d'objets vis à vis d'une de leurs propriétés est à rapprocher du $GROUPBY$ de SQL.