College of Computing

Université Mohammed VI Polytechnique

TP - Conteneurs en C++

Imad Kissami

10 Février 2025

Instructions

- L'utilisation de ChatGPT est autorisée, mais chaque ligne de code doit être comprise.
- Chaque code doit être compilé via des commandes en ligne.
- Chaque TP doit être placé dans un répertoire nommé **TP[numéro]_Nom_Prénom**.
- Chaque répertoire doit contenir un fichier **README** expliquant comment exécuter les codes.
- Le fichier principal de chaque TP doit être nommé **main.cpp**.
- S'il y a d'autres fichiers en plus du fichier principal, ils doivent être nommés **nomfichier_exo[numéro].[hpp/cpp]**.
- Les codes doivent être commentés.
- Pour ce TP il faut avoir un seul fichier **main.cpp**, qui utilise les directives du préprocesseur pour choisir quel exercice compiler. Par défaut, l'exercice 1 sera compilé, mais vous pouvez définir la macro EXO lors de la compilation pour sélectionner l'exercice souhaité (par exemple, en passant **-DEXO=2** pour l'exercice 2).
- Utiliser `auto` et `decltype` dès que possible

Exercice 1 : Manipulation d'un tableau C 1D

Problème : - Déclarer un tableau C de taille 5 et l'initialiser avec les nombres : `10, 20, 30, 40, 50`. - Utiliser une boucle pour afficher tous les éléments. - Calculer et afficher la somme des éléments.

Sortie attendue :

```
Éléments : 10 20 30 40 50
Somme : 150
```

Exercice 2 : Tableau C 2D (Stockage Matriciel)

Problème : - Déclarer une matrice `3x3` et l'initialiser. - Afficher la matrice ligne par ligne. - Calculer et afficher la somme des éléments diagonaux.

Sortie attendue :

```
Matrice :
1  2  3
4  5  6
7  8  9
Somme diagonale : 15
```

Exercice 3 : Utilisation de `std::array`

Problème : - Utiliser `std::array<int, 5>` pour stocker cinq nombres. - Inverser le tableau et afficher les éléments.

Sortie attendue :

```
Original : 1 2 3 4 5
Inversé : 5 4 3 2 1
```

Exercice 4 : Utilisation de `std::vector`

Problème : - Utiliser `std::vector<int>` pour stocker des nombres saisis par l'utilisateur. - Doubler chaque nombre et afficher le résultat.

Sortie attendue (pour entrée : 1 2 3 4 5) :

```
Doublé : 2 4 6 8 10
```

Exercice 5 : Utilisation de `std::list` (Liste Chaînée)

Problème : - Utiliser `std::list<int>` pour stocker des nombres. - Insérer un nombre au début et à la fin. - Afficher la liste finale.

Sortie attendue :

```
Après insertion : 0 10 20 30 40 50 60
```

Exercice 6 : Utilisation de `std::set` (Éléments Uniques)

Problème : - Insérer des nombres dans un `std::set`. - Essayer d'insérer des doublons et afficher le set.

Sortie attendue :

```
Original : 5 10 15 20
Après insertion de 10 : 5 10 15 20 (pas de doublons)
```

Exercice 7 : Utilisation de `std::map` (Paires Clé-Valeur)

Problème : - Stocker des noms d'étudiants et leurs notes dans un `std::map`. - Afficher le nom de chaque étudiant avec sa note.

Sortie attendue :

```
John : 85
Alice : 90
Bob : 78
```

Exercice 8 : Compteur de mots avec `std::unordered_map`

Problème : - Compter le nombre d'occurrences des mots dans une phrase donnée. - Utiliser un `std::unordered_map`.

Sortie attendue (pour l'entrée : "pomme banane pomme orange banane banane") :

```
pomme : 2
banane : 3
orange : 1
```

Exercice 9 : Trier un `std::vector`

Problème : - Stocker une liste de nombres dans un `std::vector`. - Trier les nombres en ordre croissant avec `std::sort()`.

Sortie attendue :

```
Original : 30 10 50 20 40
Trié : 10 20 30 40 50
```

Exercice 10 : Trouver la Plus Longue Séquence Consécutive dans un `std::set`

Problème : - Trouver la plus longue séquence de nombres consécutifs dans une liste donnée. - Utiliser un `std::set<int>` pour un accès rapide.

Entrée Exemple :

```
{100, 4, 200, 1, 3, 2}
```

Sortie attendue :

```
Plus longue séquence : 4 (1, 2, 3, 4)
```

Exercice 11 : Implémentation d'un Cache LRU (Least Recently Used)

Objectif : Dans cet exercice, vous allez implémenter un **cache LRU (Least Recently Used)** en utilisant :

- Une **liste doublement chaînée** (`std::list`) pour stocker les éléments en **ordre d'utilisation**.
- Une **table de hachage** (`std::unordered_map`) pour assurer un **accès rapide** aux éléments en $O(1)$.

Règles du cache LRU :

1. **Quand une nouvelle paire (clé, valeur) est insérée :**
   - Si la clé existe déjà, elle est mise à jour et déplacée en **tête** (car récemment utilisée).
   - Si la clé n'existe pas et que la capacité maximale est atteinte, l'élément **le moins récemment utilisé** est supprimé (le dernier élément de la liste).
2. **Quand une clé est accédée (`get(key)`) :**
   - Si la clé existe, elle est déplacée en **tête de liste** car elle vient d'être utilisée.
   - Si la clé n'existe pas, la fonction retourne `-1`.

Instructions :

1. **Implémentez une fonction** `put(cache, lru, capacity, key, value)` qui :
   - Insère une nouvelle clé-valeur dans le cache.
   - Déplace la clé en tête de liste si elle existe déjà.
   - Supprime l'élément **le moins récemment utilisé** si la capacité est atteinte.
2. **Implémentez une fonction** `get(cache, lru, key)` qui :
   - Retourne la valeur associée à une clé si elle existe.
   - Déplace cette clé en **tête de liste**.
   - Retourne `-1` si la clé n'est pas trouvée.
3. **Implémentez une fonction** `display(lru)` qui affiche le contenu du cache.

4. **Testez votre implémentation avec les étapes suivantes :**
   — Ajoutez `(1, 10)` et `(2, 20)` au cache.
   — Accédez à `1`, ce qui le rend récemment utilisé.
   — Ajoutez `(3, 30)`, ce qui doit supprimer `(2, 20)`, car `(1, 10)` est récemment utilisé.
   — Affichez l'état du cache après chaque opération.

Choix des conteneurs

Afin d'optimiser les performances, vous devez choisir les conteneurs adaptés pour :

— **Accès rapide aux éléments via leur clé** : utilisez `std::unordered_map` pour stocker les associations {*clé -> itérateur*} et permettre un accès en $\mathcal{O}(1)$.
— **Gestion de l'ordre des éléments en fonction de leur utilisation** : utilisez `std::list` pour stocker les clés dans l'ordre d'accès, ce qui permet de déplacer rapidement un élément en **tête** en $\mathcal{O}(1)$.
— `std::unordered_map<int, std::list<std::pair<int, int>>::iterator>`
   — Associe une **clé unique** à un **itérateur** pointant vers l'élément correspondant dans la liste.
   — Permet un accès en $\mathcal{O}(1)$ aux éléments stockés dans la liste.
— `std::list<std::pair<int, int>>`
   — Stocke les paires `(clé, valeur)` dans l'ordre d'utilisation.
   — Permet d'accéder en $\mathcal{O}(1)$ au **plus récemment utilisé** (tête) et au **moins récemment utilisé** (queue).

Exemple d'exécution attendue :

```
État du cache : (2, 20) (1, 10)
Accès à la clé 1 : 10
État du cache : (1, 10) (2, 20)
Ajout de (3, 30), suppression de (2, 20)
État du cache : (3, 30) (1, 10)
```