# Introduction aux Hashmaps en C++

March 21, 2025

# Introduction

- Les hashmaps (ou tables de hachage) permettent une recherche rapide en $O(1)$ en moyenne.
- Utilisées massivement dans les bases de données, systèmes de cache, compilateurs.
- Présentes dans la STL de C++ sous la forme de `std::unordered_map`.

# Principe des Hashmaps

- Stocke des couples (clé, valeur).
- Utilise une fonction de hachage pour déterminer l'emplacement des données.
- En cas de collision, différentes stratégies existent : chaînage, open addressing.

# Exemple de Hashmap naïve I

```cpp
#include <iostream>
#include <vector>
#include <list>

struct HashMap {
    static const int SIZE = 10;
    std::vector<std::list<std::pair<int, std::string>>> table;

    HashMap() : table(SIZE) {}

    int hash(int key) {
        return key % SIZE;
    }

    void insert(int key, std::string value) {
        int h = hash(key);
        table[h].push_back({key, value});
    }

    std::string get(int key) {
        int h = hash(key);
        for (auto &p : table[h]) {
            if (p.first == key) return p.second;
```

# Exemple de Hashmap naïve II

```
        }
        return "Not found";
    }
};
```

# Utilisation de la Hashmap naïve I

```
int main() {
    HashMap hm;
    hm.insert(1, "Alice");
    hm.insert(2, "Bob");
    hm.insert(11, "Charlie");
    std::cout << hm.get(1) << std::endl;
    std::cout << hm.get(11) << std::endl;
    std::cout << hm.get(3) << std::endl;
    return 0;
}
```

# Limites et optimisations

- Risque de collisions si la fonction de hachage est mauvaise.
- Utilisation d'open addressing ou d'autres stratégies pour réduire l'encombrement mémoire.
- Meilleures performances avec des nombres premiers pour la taille du tableau.

# Rehashing : Qu'est-ce que c'est et pourquoi est-ce important ?

- **Définition** : Le *rehashing* est le processus de redimensionnement d'une table de hachage lorsqu'elle atteint une certaine capacité, afin de maintenir une performance optimale.
- **Objectif** : Réduire le nombre de collisions et améliorer l'efficacité des opérations de recherche, d'insertion et de suppression.
- **Quand intervient-il ?** : Généralement déclenché lorsque le facteur de charge (ratio entre le nombre d'éléments et la taille de la table) dépasse un seuil prédéfini.
- **Processus** :
  1. Créer une nouvelle table de hachage avec une taille plus grande.
  2. Recalculer les indices pour chaque clé en utilisant une nouvelle fonction de hachage adaptée à la nouvelle taille.
  3. Insérer chaque élément de l'ancienne table dans la nouvelle.

# Exemple d'algorithme de hachage avec rehashing I

Considérons une fonction de hachage naïve utilisant le modulo pour déterminer l'index :

- **Fonction de hachage initiale** : $h(k) = k \mod 10$
- **Table de hachage initiale** : Taille de 10

Si la table devient trop pleine, on effectue un *rehashing* :

- **Nouvelle fonction de hachage** : $h'(k) = k \mod 20$
- **Nouvelle table de hachage** : Taille de 20

```
#include <iostream>
#include <vector>
#include <list>

struct HashMap {
    std::vector<std::list<std::pair<int,
     std::string>>> table;
    int num_elements;
    float load_factor;

    HashMap(int size = 10, float lf = 0.75) : table(size),
     num_elements(0), load_factor(lf) {}
```

# Exemple d'algorithme de hachage avec rehashing II

```
int hash(int key, int size) const {
    return key % size;
}

void insert(int key, const std::string& value) {
    if (1. * num_elements / table.size() >= load_factor) {
        rehash();
    }
    int h = hash(key, table.size());
    table[h].push_back({key, value});
    ++num_elements;
}

void rehash() {
    int new_size = table.size() * 2;
    std::vector<std::list<std::pair<int, std::string>>>
    new_table(new_size);

    for (const auto& bucket : table) {
        for (const auto& p : bucket) {
            int h = hash(p.first, new_size);
            new_table[h].push_back(p);
        }
```

# Exemple d'algorithme de hachage avec rehashing III

```
            }
            table = std::move(new_table);
        }
};
```

Cet exemple montre comment les éléments sont réinsérés dans une table de taille double, réduisant ainsi le facteur de charge et les collisions.

# Explication de `std::move`

- **Définition** : `std::move` est une fonction qui convertit un objet en une valeur *rvalue*, permettant ainsi le déplacement des ressources au lieu d'une copie.
- **Pourquoi l'utiliser ?**
  - Évite une copie coûteuse des objets lourds comme les vecteurs ou les listes.
  - Améliore les performances en transférant directement la mémoire sans duplication.
  - Laisse l'objet source dans un état valide mais indéfini.
- **Exemple dans notre code** :

```
table = std::move(new_table);
```

Cela permet d'affecter directement les ressources de `new_table` à `table`, sans recréer et recopier tous les éléments.

# Utilisation de std::unordered_map I

```cpp
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<int, std::string> hashmap;
    hashmap[1] = "Alice";
    hashmap[2] = "Bob";
    hashmap[11] = "Charlie";

    std::cout << hashmap[1] << std::endl;
    std::cout << hashmap[11] << std::endl;
    std::cout << hashmap[3] << std::endl; // Affiche une chaine vi
    return 0;
}
```

# Pourquoi le choix d'une bonne fonction de hachage est crucial ?

- Une mauvaise fonction de hachage entraîne un grand nombre de collisions, réduisant ainsi l'efficacité des opérations de recherche et d'insertion.
- Une bonne fonction doit être rapide à calculer et générer une distribution uniforme des clés pour éviter les regroupements (clustering).
- L'impact des collisions est encore plus important dans les environnements haute performance comme les bases de données et les systèmes distribués.

# Exemples de fonctions de hachage utilisées

- **MurmurHash** : Très performant pour les chaînes de caractères, utilisé dans de nombreux systèmes comme LevelDB.
- **CityHash** : Optimisé pour les architectures modernes, utilisé par Google.
- **xxHash** : Ultra-rapide, efficace pour le traitement de gros volumes de données.
- **FNV-1a** : Simple et efficace pour de petits ensembles de données.
- **SipHash** : Sécurisé contre certaines attaques, souvent utilisé dans les structures de hachage publiques.

# Fonction de hachage FNV-1a : Aperçu

- **Origine** : Développée par Glenn Fowler, Landon Curt Noll et Kiem-Phong Vo.
- **Principe** : Utilise des opérations simples (XOR et multiplication) pour générer des valeurs de hachage.
- **Variantes** : Disponible en versions 32, 64, 128, 256, 512 et 1024 bits.

# Pourquoi FNV-1a est-elle intéressante ?

- **Simplicité** : Facile à implémenter avec des opérations de base.
- **Rapidité** : Performances élevées, adaptée aux applications nécessitant un traitement rapide.
- **Bonne dispersion** : Réduit les collisions, assurant une distribution uniforme des valeurs de hachage.
- **Polyvalence** : Efficace pour des données similaires, comme les chaînes de caractères proches (e.g., URLs, noms de fichiers).

# Fonction de hachage FNV-1a : Formule

- **Initialisation** :

$$\text{hash} \leftarrow \text{offset\_basis}$$

- **Pour chaque octet du `data` à hacher** :
  1. Appliquer l'opération XOR entre le hash courant et l'octet :

$$\text{hash} \leftarrow \text{hash} \oplus \text{octet}$$

  2. Multiplier le résultat par une constante première :

$$\text{hash} \leftarrow \text{hash} \times \text{FNV\_prime}$$

- **Valeurs typiques** :
  - `FNV_prime` (32 bits) : 16777619
  - `offset_basis` (32 bits) : 2166136261

# Implémentation de `std::unordered_map`

- Basée sur une table de hachage utilisant le chaînage séparé (chaque bucket contient une liste chaînée d'éléments).
- Offre une complexité moyenne en temps constant $O(1)$ pour les opérations d'insertion, de suppression et de recherche.
- Utilise par défaut la fonction de hachage `std::hash`, personnalisable selon les besoins.

# Alternatives optimisées : `absl::flat_hash_map`

- Développée par Google, cette implémentation stocke directement les paires clé-valeur dans un tableau, réduisant l'indirection mémoire.
- Utilise des techniques avancées pour améliorer l'utilisation du cache CPU et réduire les collisions.
- Offre des performances supérieures en termes de vitesse et d'efficacité mémoire par rapport à `std::unordered_map`.

# Conclusion

- Les tables de hachage sont essentielles en informatique moderne.
- Le choix d'une bonne fonction de hachage est crucial.
- La STL offre `std::unordered_map`, une solution rapide et optimisée.

# Optimisation : L'algorithme Suisse de Google

- Développé par Google pour une meilleure gestion des tables de hachage, basé sur une approche hybride entre hachage et structures d'indexation avancées.
- Optimise l'utilisation du cache CPU et réduit les collisions grâce à un contrôle fin des métadonnées.
- Implémenté dans `absl::flat_hash_map` et `absl::node_hash_map`, qui surpassent `std::unordered_map` en termes de vitesse et d'efficacité mémoire.

# Pourquoi optimiser les tables de hachage ?

- Les performances des tables de hachage dépendent fortement des accès mémoire et des collisions.
- L'optimisation permet d'améliorer la rapidité des requêtes tout en réduisant l'utilisation mémoire.
- L'algorithme Suisse de Google a été conçu pour répondre aux besoins des applications haute performance.

# Comparaison avec `std::unordered_map`

- `std::unordered_map` utilise des listes chaînées pour gérer les collisions, ce qui entraîne un grand nombre d'allocation mémoire.
- L'algorithme Suisse privilégie une approche en tableau compact, améliorant ainsi l'efficacité du cache CPU.
- Les benchmarks montrent un gain de vitesse significatif pour l'insertion et la recherche.

# Technique 1 : Probe Sequence optimisée et SIMD

- **Probing** : technique de recherche d'une position libre lors de l'insertion d'un élément.
- Utilisation du probing linéaire avec un balayage en bloc pour minimiser les accès mémoire incohérents.
- Exploitation des instructions SIMD (Single Instruction, Multiple Data) pour tester plusieurs emplacements en parallèle, accélérant ainsi la détection de clés vides ou occupées.
- Réduction du taux de cache misses en regroupant les accès mémoire et en évitant les **rehash** trop fréquents.

# Probing linéaire vs. Probing quadratique

- **Probing linéaire** : recherche d'une position libre en avançant de manière séquentielle.
- **Probing quadratique** : saut exponentiel pour minimiser certaines collisions, mais avec des accès mémoire moins efficaces.
- L'algorithme Suisse privilégie un compromis optimal entre les deux pour améliorer la performance.

# Qu'est-ce que le rehashing ?

- Lorsque la table de hachage atteint un certain taux d'occupation, elle est redimensionnée et toutes les clés sont réinsérées.
- Cette opération est coûteuse en temps de calcul et en mémoire.
- L'algorithme Suisse minimise les rehashs en optimisant l'utilisation de l'espace et en réduisant le taux de collision.

# Qu'est-ce qu'un bucket ?

- Un **bucket** est une case mémoire dans la table de hachage qui peut contenir une clé et sa valeur associée.
- En fonction de l'implémentation, un bucket peut contenir un seul élément ou plusieurs (dans le cas de listes chaînées ou d'autres stratégies de collision).
- L'algorithme Suisse optimise la gestion des buckets en réduisant leur occupation mémoire et en maximisant l'efficacité du cache CPU.

# Utilisation des instructions SIMD

- SIMD permet d'exécuter plusieurs opérations de comparaison simultanément.
- Réduction des accès mémoire aléatoires et amélioration de la vitesse de recherche.
- Implémenté avec `AVX2` ou `SSE` selon l'architecture CPU.

# Technique 2 : Small Object Optimization et Control Bits

- Stockage compact des métadonnées à l'aide de **control bits**, permettant une gestion efficace des entrées sans surcharge mémoire excessive.
- Implémentation de **SOO (Small Object Optimization)**, qui évite des allocations dynamiques inutiles pour les objets de petite taille, réduisant ainsi la fragmentation mémoire.
- Utilisation d'un modèle de **packed metadata**, où chaque bucket contient des informations supplémentaires permettant d'optimiser les insertions et suppressions.

# Les Control Bits en détail

- Chaque bucket contient des bits de contrôle indiquant l'état de l'entrée (occupé, libre, supprimé).
- Cela permet un parcours plus rapide lors de la recherche d'une clé.
- Optimisation de l'utilisation du cache et des sauts conditionnels.

# Optimisation mémoire avec SOO

- Réduction des allocations mémoire pour les objets de petite taille.
- Moins de fragmentation et meilleur usage du cache CPU.
- Implémentation efficace dans `absl::flat_hash_map`.