Gestion de l'allocation mémoire en C++

Exemple d'une classe `Vector` et gestion manuelle

L3 maths Strasbourg

14 février 2025

Plan de la présentation

Enjeux de l'allocation mémoire en C++

- **Allocation dynamique** via `new` et `delete`.
- Risques de **fuites de mémoire** et de **fragmentation**.
- Nécessité d'une gestion rigoureuse pour éviter les erreurs et optimiser les performances.
- Comparaison avec des langages à **ramasse-miettes automatique**.

Le ramasse-miettes (Garbage Collector)

- Le **ramasse-miettes** est un mécanisme qui automatise la libération de la mémoire non utilisée.
- Il identifie les objets qui ne sont plus accessibles et les détruit pour libérer la mémoire.
- Il existe le Boehm GC, un ramasse-miettes pour le C++, mais avec des limitations.
- Certains langages (Java, C#) utilisent des GC avancés pour simplifier la programmation.
- Cette approche présente l'inconvénient de **coûts en performances** et de **prédictibilité limitée**.

Le comptage de références pour la gestion mémoire

- Technique consistant à compter le nombre de références pointant vers un même objet.
- Lorsqu'une copie est effectuée, le compteur est incrémenté.
- À la destruction d'une instance, le compteur est décrémenté.
- Lorsque le compteur atteint zéro, la mémoire occupée est libérée.
- Attention aux **constructeurs cachés** et à la gestion des copies implicites qui peuvent fausser le comptage.

Présentation de l'exemple

- Nous allons illustrer la gestion manuelle de la mémoire à travers une classe `Vector`.
- Cette classe gère un tableau dynamique d'entiers avec un système de comptage de références.
- L'exemple aborde :
  - La construction et la copie de l'objet.
  - La méthode `push_back` avec gestion du partage et de la copie sur écriture.
  - La libération de la mémoire via le décrément du compteur.

Implémentation de la classe `Vector` I

Listing 1 – Classe Vector avec comptage de références

```
#include <iostream>

class Vector {
private:
    int* data;
    size_t size;
    size_t capacity;
    unsigned* refCount;

    // Libération de la mémoire partagée
    void release() {
        if(--(*refCount) == 0) {
            delete[] data;
            delete refCount;
        }
    }

public:
    // Constructeur avec capacité initiale (par défaut 10)
    Vector(size_t cap = 10)
        : size(0), capacity(cap), refCount(new unsigned(1)) {
        data = new int[capacity];
    }

    // Constructeur de copie : partage les données et incrémente le compteur
    Vector(const Vector& other)
        : data(other.data), size(other.size), capacity(other.capacity),
          refCount(other.refCount) {
        ++(*refCount);
```

Implémentation de la classe `Vector` II

```
    }

    // Opérateur d'affectation : gère le partage et le compteur
    Vector& operator=(const Vector& other) {
        if (this != &other) {
            release();
            data = other.data;
            size = other.size;
            capacity = other.capacity;
            refCount = other.refCount;
            ++(*refCount);
        }
        return *this;
    }

    // Destructeur : décrémente le compteur et libère la mémoire si nécessaire
    ~Vector() {
        release();
    }

    // Ajout d'un élément à la fin du vecteur
    void push_back(int value) {
        // Si le vecteur est partagé, effectuer une copie pour éviter
        // des effets de bord (copie sur écriture)
        if(*refCount > 1) {
            int* newData = new int[capacity];
            for(size_t i = 0; i < size; ++i)
                newData[i] = data[i];
            --(*refCount);
            data = newData;
            refCount = new unsigned(1);
        }
```

Implémentation de la classe `Vector` III

```
        // Agrandissement du tableau si nécessaire
        if(size == capacity) {
            capacity *= 2;
            int* newData = new int[capacity];
            for(size_t i = 0; i < size; ++i)
                newData[i] = data[i];
            delete[] data;
            data = newData;
        }
        data[size++] = value;
    }

    // Affichage des éléments du vecteur
    void print() const {
        for(size_t i = 0; i < size; ++i)
            std::cout << data[i] << " ";
        std::cout << std::endl;
    }
};

int main() {
    Vector v1;
    v1.push_back(5);
    v1.push_back(10);

    // Copie de v1 : partage des données grâce au comptage de références
    Vector v2 = v1;
    v2.push_back(15); // Déclenche une copie sur écriture

    std::cout << "v1 : ";
    v1.print();
    std::cout << "v2 : ";
```

Implémentation de la classe `Vector` IV

```
    v2.print();

    return 0;
}
```

Analyse de l'exemple

- **Constructeur de copie** : partage les mêmes données et incrémente `refCount`.
- **Opérateur d'affectation** : assure le même comportement en évitant l'auto-affectation.
- **Méthode `push_back`** :
  - Vérifie si le vecteur est partagé (`*refCount > 1`).
  - En cas de partage, réalise une **copie sur écriture** pour isoler la modification.
  - Gère l'agrandissement du tableau en doublant la capacité.
- **Destructeur** : décrémente le compteur et libère la mémoire si aucun autre objet ne partage les données.

Points de vigilance

- La gestion manuelle du comptage de références est source d'erreurs :
  - Risque de **fuites mémoire** si le compteur n'est pas correctement décrémenté.
  - **Constructeurs cachés** et copies implicites pouvant corrompre le compteur.
- La copie sur écriture permet d'éviter des modifications imprévues sur des données partagées, mais complexifie la logique.
- Ce modèle est simplifié et ne gère pas tous les cas (par exemple, la concurrence ou la constance stricte).

Conclusion et perspectives

- La gestion manuelle de l'allocation mémoire en C++ nécessite rigueur et prudence.
- Le comptage de références est une technique utile pour le partage de ressources, mais elle est fragile.
- Le mécanisme de ramasse-miettes, bien que non standard en C++, offre une approche alternative dans d'autres langages.
- Le prochain cours abordera les **smart pointers** pour simplifier cette gestion.

Questions et discussion

Des questions ?