Utilisation des Pointeurs Intelligents en C++

L3 maths Strasbourg

14 février 2025

Plan de la présentation

Introduction aux Pointeurs Intelligents

Simplification d'un Exemple : La classe Vector

Conclusion

Pourquoi utiliser des pointeurs intelligents ?

- Éliminer la gestion manuelle de la mémoire (allocation, libération, gestion des compteurs).
- Réduire les risques de fuites mémoire et d'erreurs liées à la gestion de ressources.
- Simplifier le code et améliorer sa robustesse.
- Gérer automatiquement la durée de vie des objets.

Les principaux types de pointeurs intelligents

- **`std::unique_ptr`** : propriété exclusive, pas de partage.
- **`std::shared_ptr`** : partage de la propriété avec un comptage de références automatique.
- **`std::weak_ptr`** : référence non propriétaire pour éviter les cycles.

Comparaison avec l'approche manuelle

- Dans l'exemple manuel, la gestion du compteur de références et de l'allocation/désallocation du tableau était complexe.
- Les pointeurs intelligents remplacent la logique de gestion manuelle par une solution standardisée.
- Le code devient plus lisible et moins sujet aux erreurs.

Exemple de classe `Vector` sans pointeurs intelligents

Listing 1 – Version manuelle avec comptage de références

```
class Vector {
private:
    int* data;
    size_t size;
    size_t capacity;
    unsigned* refCount;
    // Méthode release() pour libérer la mémoire si nécessaire
public:
    Vector(size_t cap = 10) : size(0), capacity(cap), refCount(new unsigned(1)) {
        data = new int[capacity];
    }
    // Constructeur, opérateur=, destructeur, push_back, etc.
};
```

Les avantages des pointeurs intelligents

- Gestion automatique de la durée de vie.
- Plus besoin d'écrire du code redondant pour le comptage de références.
- Moins de risques d'oublier de libérer la mémoire.
- Intégration avec les fonctionnalités modernes du C++ (RAII, exceptions sécurisées).

Classe Vector avec un shared pointer I

```
    #include <iostream>
#include <memory>

class Vector
{
private:
    std::shared_ptr<int[]> data; // Allocation automatique de l'array
    size_t size;
    size_t capacity;

public:
    // Constructeur avec capacité par défaut (10)
    Vector(size_t cap = 10)
        : data(new int[cap], std::default_delete<int[]>()),
          size(0), capacity(cap) {}

    // Méthode d'ajout d'un élément
    void push_back(int value)
    {
        // Si le vecteur est plein, doubler la capacité
        if (size == capacity)
        {
            capacity *= 2;
            std::shared_ptr<int[]> newData(
                new int[capacity], std::default_delete<int[]>());
            for (size_t i = 0; i < size; ++i)
            {
                newData[i] = data[i];
            }
            data = newData;
        }
        data[size++] = value;
```

Classe Vector avec un shared pointer II

```
    }

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

int main()
{
    Vector v; // Utilise la capacité par défaut de 10
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.print(); // Affiche: 1 2 3
    return 0;
}
```

Analyse de la version avec pointeurs intelligents

- **Allocation simplifiée** : La mémoire du tableau est allouée via `std::shared_ptr<int[]>`.
- **Pas de gestion manuelle du compteur** : Le comptage de références est géré automatiquement.
- **Libération automatique** : La mémoire est libérée dès qu'aucune instance ne la référence.
- **Lisibilité améliorée** : Le code est plus court et plus facile à maintenir.

Comparaison entre les deux approches

- **Approche manuelle** :
  - Plus de code à écrire et à maintenir.
  - Risque d'erreurs dans la gestion de la mémoire.
- **Avec pointeurs intelligents** :
  - Code simplifié et sécurisé.
  - Moins de risques de fuites mémoire et d'erreurs de comptage.

Conclusion et Perspectives

- Les pointeurs intelligents offrent une gestion automatique et sécurisée de la mémoire.
- L'utilisation de `std::shared_ptr` dans notre exemple a simplifié la classe `Vector`.
- Adopter les pointeurs intelligents permet de se concentrer sur la logique métier plutôt que sur la gestion de la mémoire.
- Ils sont indispensables pour écrire du code C++ moderne et robuste.

Questions et Discussion

Des questions ?