Templates en C++

L3 math Informatique S6

26 février 2025

Introduction aux Templates

- Les templates permettent d'écrire du code générique.
- Ils sont utilisés pour créer des fonctions et des classes réutilisables.
- Exemple simple : un vecteur de flottants de taille deux avec addition.

Principe des Templates

- Les templates permettent de définir des fonctions ou des classes sans spécifier le type de données.
- Le type est déterminé lors de l'instanciation.

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

La plupart du temps on peut remplacer `typename` par `class`.

Exemple de Template de Fonction

- Utilisation d'un template pour additionner deux nombres.

```
int resultInt = add(3, 4);
double resultDouble = add(3.5, 2.5);
```

Classes Templates

- Les classes templates permettent de créer des structures de données génériques.

```
template <typename T>
class Vector2 {
public:
    T x, y;
    Vector2(T x, T y) : x(x), y(y) {}
    Vector2 operator+(const Vector2& other) {
        return Vector2(x + other.x, y + other.y);
    }
};
```

Utilisation de Classes Templates

- Instanciation et utilisation de la classe template `Vector2`.

```
Vector2<int> vecInt(1, 2);
Vector2<double> vecDouble(1.5, 2.5);
Vector2<int> sumInt = vecInt + vecInt;
```

Types Paramétrés

- Les types paramétrés permettent de définir des types génériques.
- Ils sont spécifiés lors de l'instanciation du template.

Aspects Pratiques

- Les templates sont généralement définis dans des fichiers d'en-tête (.hpp).
- Pas de fichiers .cpp séparés pour les templates.
- Tout le code doit être visible au compilateur.

Exemple Complet : Vecteur de Flottants

```cpp
#include <iostream>

template <typename T>
class Vector2 {
public:
    T x, y;
    Vector2(T x, T y) : x(x), y(y) {}
    Vector2 operator+(const Vector2& other) {
        return Vector2(x + other.x, y + other.y);
    }
};

int main() {
    Vector2<int> vecInt(1, 2);
    Vector2<double> vecDouble(1.5, 2.5);
    Vector2<int> sumInt = vecInt + vecInt;
    std::cout << "Sum: " << sumInt.x << ", " << sumInt.y << std::endl;
    return 0;
}
```

Classe Template avec Deux Types

- Une classe template peut dépendre de plusieurs types.
- Exemple : une classe `Pair` qui stocke deux valeurs de types différents.

```
template <typename T1, typename T2>
class Pair {
public:
    T1 first;
    T2 second;
    Pair(T1 first, T2 second) : first(first), second(second) {}
    void display() {
        std::cout << "First: " << first << ", Second: " << second << std::endl;
    }
};
```

Utilisation de la Classe `Pair`

- Instanciation et utilisation de la classe `Pair` avec différents types.

```
Pair<int, double> p1(1, 2.5);
Pair<std::string, char> p2("Hello", 'A');
p1.display();
p2.display();
```

Classe Imbriquée avec Templates

- Une classe template peut contenir une autre classe template imbriquée.
- Exemple : une classe `Outer` contenant une classe imbriquée `Inner`.

```
template <typename T>
class Outer {
public:
    class Inner {
    public:
        T value;
        Inner(T value) : value(value) {}
        void display() {
            std::cout << "Inner value: " << value << std::endl;
        }
    };
};
```

Problématiques de Compilation avec les Templates

- **Compilation séparée :** Les templates doivent être définis dans les fichiers d'en-tête (.hpp) car le compilateur a besoin de voir la définition complète du template.
- **Erreurs de compilation :** Les erreurs liées aux templates peuvent être difficiles à diagnostiquer en raison de la complexité des messages d'erreur générés.
- **Temps de compilation :** L'utilisation intensive de templates peut augmenter le temps de compilation, car le compilateur doit instancier le code pour chaque type utilisé.
- **Compatibilité :** Les templates peuvent poser des problèmes de compatibilité entre différents compilateurs ou versions de compilateurs.
- **Débogage :** Le débogage des templates peut être complexe, car les types ne sont pas connus avant l'instanciation.