INTRODUCTION AUX TEMPLATES

DÉFINITION

Les templates en C++ permettent d'écrire du code générique. Ils permettent de créer des fonctions et des classes qui peuvent fonctionner avec n'importe quel type de données. Les templates sont utilisés pour réduire la redondance du code. Ils permettent d'écrire des algorithmes indépendants du type. Les templates sont définis à la compilation, ce qui améliore les performances.

POURQUOI UTILISER DES TEMPLATES

Réduction de la redondance du code. Facilite la maintenance et l'extension du code. Permet de créer des algorithmes génériques. Améliore les performances grâce à la génération de code spécifique. Encourage la réutilisation du code. Simplifie la gestion des types dans les bibliothèques.

SYNTAXE DE BASE

La syntaxe de base d'un template est :

```
template <typename T>
T function(T arg) {
    // Code
}
```

`typename` peut être remplacé par `class`. Les templates peuvent avoir plusieurs paramètres de type.

TEMPLATES DE FONCTIONS

Les templates de fonctions permettent de créer des fonctions génériques.

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

Ils peuvent être utilisés avec différents types de données.

TEMPLATES DE CLASSES

Les templates de classes permettent de créer des classes génériques.

```
template <typename T>
class MyClass {
    T data;
public:
    MyClass(T arg) : data(arg) {}
    T getData() { return data; }
};
```

Ils permettent de créer des structures de données génériques.

PARAMÈTRES DE TEMPLATES

Les templates peuvent avoir plusieurs paramètres de type.

```
template <typename T, typename U>
class MyClass {
    T data1;
    U data2;
public:
    MyClass(T arg1, U arg2) : data1(arg1), data2(arg2) {}
};
```

Les paramètres peuvent être des types ou des valeurs constantes.

SPÉCIALISATION DES TEMPLATES

La spécialisation des templates permet de définir des comportements spécifiques pour certains types.

```
template <>
class MyClass<int> {
    int data;
public:
    MyClass(int arg) : data(arg) {}
    int getData() { return data * 2; }
};
```

Elle permet d'optimiser le code pour des cas particuliers.

EXEMPLES PRATIQUES

Exemple de template de fonction :

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

Exemple de template de classe :

```
template <typename T>
class Stack {
    std::vector<T> elems;
public:
    void push(T const& elem) { elems.push_back(elem); }
    void pop() { elems.pop_back(); }
    T top() const { return elems.back(); }
};
```

LIMITATIONS ET PRÉCAUTIONS

Les templates peuvent augmenter le temps de compilation. Ils peuvent rendre le code plus complexe à comprendre. La spécialisation excessive peut compliquer la maintenance. Les erreurs de compilation peuvent être difficiles à déboguer. Utiliser avec parcimonie pour éviter la surcharge de code.

AVANTAGES DES TEMPLATES

RÉUTILISATION DU CODE

Les templates permettent de créer des fonctions et des classes génériques. Cela évite la duplication de code pour différents types de données. Une seule définition de fonction ou de classe peut être utilisée avec différents types. Cela améliore la lisibilité et la maintenabilité du code. Les templates augmentent la modularité du code.

FLEXIBILITÉ

Les templates offrent une grande flexibilité dans la programmation. Ils permettent de créer des algorithmes indépendants des types de données. Les mêmes algorithmes peuvent être appliqués à différents types de données. Cela rend le code plus adaptable aux changements de spécifications.

PERFORMANCE

L'utilisation des templates peut améliorer les performances. Le code généré par les templates est souvent aussi efficace que le code écrit manuellement. Le compilateur peut optimiser le code pour chaque type spécifique. Cela permet d'obtenir des performances proches du code spécialisé.

SÉCURITÉ DE TYPE

Les templates offrent une sécurité de type renforcée. Les erreurs de type sont détectées à la compilation plutôt qu'à l'exécution. Cela réduit les bugs et les erreurs de type. Les templates garantissent que les fonctions et les classes sont utilisées correctement.

MAINTENANCE FACILITÉE

Les templates simplifient la maintenance du code. Les modifications apportées à la définition du template se propagent automatiquement. Il n'est pas nécessaire de modifier chaque instance du code. Cela réduit le risque d'erreurs lors des mises à jour.

GÉNÉRICITÉ

Les templates permettent de créer des composants génériques. Ces composants peuvent être réutilisés dans différents contextes. Ils éliminent la nécessité de réécrire du code pour des types spécifiques. Cela conduit à un développement plus rapide et plus efficace.

OPTIMISATION PAR LE COMPILATEUR

Le compilateur C++ peut optimiser le code généré par les templates. Il peut effectuer des optimisations spécifiques au type utilisé. Cela permet d'obtenir un code plus performant et plus efficace. Les optimisations sont réalisées au moment de la compilation.

SYNTAXE DES TEMPLATES

DEFINITION

Les templates en C++ permettent de créer des fonctions et des classes génériques. Ils permettent de travailler avec des types génériques sans spécifier le type exact. Les templates sont utiles pour écrire du code réutilisable et flexible. Ils sont principalement utilisés avec les fonctions et les classes.

SYNTAXE DE BASE

La syntaxe de base pour déclarer un template est :

```
template <typename T>
```

Ici, T est un paramètre de type générique. Les templates peuvent être utilisés pour les fonctions et les classes.

PARAMÈTRES DE TYPE

Les templates peuvent avoir plusieurs paramètres de type :

```
template <typename T, typename U>
```

Les paramètres de type peuvent être des classes, des fonctions ou des variables. Ils permettent de créer des structures de données et des algorithmes génériques.

UTILISATION DES TEMPLATES

Pour utiliser un template, vous devez spécifier le type lors de l'appel :

```
templateFunction<int>(10);
```

Les compilateurs modernes peuvent souvent déduire le type automatiquement. Les templates sont instanciés avec des types spécifiques lors de la compilation.

EXEMPLES DE SYNTAXE

Déclaration d'une fonction template :

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

Déclaration d'une classe template :

```
template <typename T>
class MyClass {
    T data;
};
```

SPÉCIALISATION DES TEMPLATES

La spécialisation permet de définir des comportements spécifiques pour certains types :

```
template <>
class MyClass<int> {
    int data;
};
```

La spécialisation complète et partielle est possible. Elle est utilisée pour optimiser ou modifier le comportement pour certains types.

LIMITATIONS ET ERREURS COURANTES

Les templates peuvent augmenter le temps de compilation. Les messages d'erreur peuvent être difficiles à comprendre. Les templates ne peuvent pas être séparés en fichiers .cpp et .h facilement. Certaines fonctionnalités ne sont pas compatibles avec les templates.

TEMPLATES DE FONCTIONS

DÉFINITION

Les templates de fonctions permettent de créer des fonctions génériques. Ils permettent de définir une seule fonction pour différents types de données. Cela favorise la réutilisabilité et réduit la duplication de code.

AVANTAGES DES TEMPLATES DE FONCTIONS

- Réduction de la duplication de code
- Flexibilité accrue
- Maintenance plus facile
- Adaptabilité à différents types de données

SYNTAXE DE BASE

La syntaxe de base pour déclarer un template de fonction est :

```
template <typename T>
T nom_de_la_fonction(T parametre);
```

EXEMPLE SIMPLE

Voici un exemple simple d'un template de fonction pour trouver le maximum de deux valeurs :

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

SPÉCIALISATION DES TEMPLATES DE FONCTIONS

La spécialisation permet de définir un comportement spécifique pour un type particulier :

```
template <>
int max<int>(int a, int b) {
    return (a > b) ? a : b;
}
```

UTILISATION AVEC DIFFÉRENTS TYPES DE DONNÉES

Les templates de fonctions peuvent être utilisés avec différents types de données :

```
int a = max(3, 5);           // Utilise int
double b = max(3.2, 5.1);    // Utilise double
char c = max('a', 'z');      // Utilise char
```

LIMITATIONS ET ERREURS COURANTES

- Les templates ne peuvent pas être séparés en fichiers .h et .cpp
- Les erreurs de compilation peuvent être difficiles à déboguer
- Les templates ne peuvent pas être partiellement spécialisés

TEMPLATES DE CLASSES

DÉFINITION

Les templates de classes en C++ permettent de créer des classes génériques. Ils permettent de définir des classes indépendamment des types de données. Un template de classe est une "recette" pour créer des classes. Les types de données spécifiques sont fournis lors de l'instanciation. Les templates de classes sont utiles pour réutiliser du code.

SYNTAXE

La syntaxe de base pour déclarer un template de classe est :

```
template <typename T>
class NomClasse {
    T membre;
public:
    void fonction(T param);
};
```

AVANTAGES

- Réutilisation du code pour différents types de données.
- Facilite la maintenance et les mises à jour.
- Réduit la duplication de code.
- Permet une plus grande flexibilité et généralisation.
- Améliore la lisibilité et la clarté du code.

EXEMPLE BASIQUE

Un exemple simple de template de classe :

```
template <typename T>
class Boite {
    T contenu;
public:
    void setContenu(T c) { contenu = c; }
    T getContenu() { return contenu; }
};
```

SPÉCIALISATION DE CLASSE TEMPLATE

La spécialisation permet de définir un comportement spécifique pour un type particulier :

```
template <>
class Boite<int> {
    int contenu;
public:
    void setContenu(int c) { contenu = c; }
    int getContenu() { return contenu; }
};
```

UTILISATION AVEC TYPES PRIMITIFS

Les templates de classes peuvent être utilisés avec des types primitifs :

```
Boite<int> boiteInt;
boiteInt.setContenu(123);

Boite<double> boiteDouble;
boiteDouble.setContenu(45.67);
```

UTILISATION AVEC TYPES PERSONNALISÉS

Les templates de classes peuvent aussi être utilisés avec des types personnalisés :

```
class Personne {
    std::string nom;
public:
    Personne(std::string n) : nom(n) {}
};

Boite<Personne> boitePersonne;
boitePersonne.setContenu(Personne("Alice"));
```

LIMITATIONS

- Les templates peuvent augmenter la complexité du code.
- La compilation peut être plus lente avec des templates.
- Les messages d'erreur peuvent être difficiles à comprendre.
- Pas de support direct pour des types spécifiques comme les pointeurs de membres.
- Les templates ne peuvent pas être séparés en fichiers .h et .cpp de manière traditionnelle.

COMPARAISON AVEC TEMPLATES DE FONCTIONS

- Les templates de fonctions définissent des fonctions génériques.
- Les templates de classes définissent des classes génériques.
- Les templates de fonctions sont plus simples à utiliser pour des opérations simples.
- Les templates de classes sont plus adaptés pour des structures de données complexes.
- Les deux permettent une réutilisation du code et une plus grande flexibilité.

PARAMÈTRES DE TEMPLATES

DÉFINITION

Les paramètres de templates permettent de généraliser les fonctions et les classes. Ils permettent de créer du code générique, réutilisable pour différents types. Les templates en C++ peuvent avoir des paramètres de type, non-type ou template template.

SYNTAXE DES PARAMÈTRES DE TEMPLATES

La syntaxe de base pour déclarer des paramètres de templates est :

```
template <typename T>
class MyClass {
    // Code de la classe
};
```

TYPES DE PARAMÈTRES DE TEMPLATES

Les types de paramètres de templates en C++ incluent :

- Paramètres de type
- Paramètres non-type
- Paramètres de template template

PARAMÈTRES DE TYPE

Les paramètres de type spécifient le type de données à utiliser. Par exemple, `typename T` ou `class T` :

```
template <typename T>
void func(T arg) {
    // Code de la fonction
}
```

PARAMÈTRES NON-TYPE

Les paramètres non-type spécifient une valeur constante. Par exemple, un entier ou un pointeur :

```
template <int N>
class Array {
    int arr[N];
};
```

PARAMÈTRES DE TEMPLATE TEMPLATE

Les paramètres de template template permettent de passer un template comme paramètre. Par exemple :

```
template <template <typename> class Container>
class Wrapper {
    Container<int> data;
};
```

UTILISATION DES PARAMÈTRES DE TEMPLATES

Les paramètres de templates sont utilisés pour créer des classes et fonctions génériques. Ils permettent de manipuler différents types sans dupliquer le code.

EXEMPLES PRATIQUES

Exemple avec un paramètre de type :

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

Exemple avec un paramètre non-type :

```
template <int N>
class FixedArray {
    int data[N];
};
```

LIMITATIONS ET CONTRAINTES

- Les paramètres non-type doivent être des valeurs constantes.
- Les templates peuvent augmenter la complexité du code.
- Les erreurs de compilation peuvent être difficiles à comprendre.