Templates en C++

Introduction aux templates

Qu'est-ce qu'un template ?

Définition

Un **template** est un modèle générique qui permet de créer des **fonctions** ou des **classes polymorphes** en C++.

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

int main() {
  int entier1 = 10;
  int entier2 = 20;
  double flottant1 = 15.5;
  double flottant2 = 7.5;

  cout << "Somme entiers: " << somme(entier1, entier2) << endl;
  cout << "Somme flottants: " << somme(flottant1, flottant2) << endl;

  return 0;
}
```

Les templates en C++ permettent d'écrire un code plus réutilisable et générique pour les fonctions ou les classes qui peuvent travailler avec différents types de données.

Utilité

Les **templates** permettent d'écrire du code **réutilisable** et **type-safe** pour différentes classes ou types de données.

Avantages	Exemples
Réutilisabilité	Éviter la duplication de code pour des classes similaires
Sécurité des types	Les erreurs de type sont détectées à la compilation plutôt qu'à l'exécution

Terminologie

- **Template** : modèle générique pour créer des fonctions ou classes polymorphes.
- **Classe template** : classe paramétrée par un ou plusieurs types.
- **Fonction template** : fonction paramétrée par un ou plusieurs types.
- **Instance de template** : version concrète d'une classe ou fonction template pour un type spécifique.

Templates de fonctions

Syntaxe

Déclaration

Pour déclarer une **fonction template**, utilisez le mot-clé `template` suivi des paramètres de type :

```
template<typename T>
T maFonction(T a, T b);
```

Implémentation

Définissez la **fonction template** comme une fonction normale, en utilisant les **paramètres de type** :

```
template<typename T>
T maFonction(T a, T b) {
  // Code de la fonction
}
```

Spécialisation de templates

Pourquoi spécialiser ?

La **spécialisation** de templates de fonctions permet de créer une version spécifique d'une fonction pour un **type particulier**.

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

template <>
const char* maxValue<const char*>(const char* a, const char* b) {
  return (strcmp(a, b) > 0) ? a : b;
}
```

La spécialisation de templates permet d'optimiser ou d'adapter le code pour des types spécifiques et d'éviter des erreurs ou des comportements indésirables.

Syntaxe

Pour **spécialiser** une fonction template, utilisez la syntaxe suivante :

```
template<>
ReturnType maFonction<Type>(Type a, Type b) {
    // Corps de la fonction spécialisée
}
```

La spécialisation permet de définir une version spécifique pour un certain type de données dans une fonction template, afin d'améliorer l'efficacité ou d'adapter le comportement pour ce type en particulier.

Exemples d'utilisation

La **spécialisation de templates** peut être utilisée pour créer des fonctions spécifiques pour un type donné :

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

Templates de classes

Templates de classes

Les **templates de classes** permettent de définir des **classes génériques**, c'est-à-dire des classes qui peuvent fonctionner avec différents types de données.

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

Pour utiliser un template de classe, il suffit d'indiquer le type de données à utiliser entre les chevrons, comme ceci :

```
MyContainer<int> containerInt;
MyContainer<double> containerDouble;
```

Syntaxe - Déclaration

Pour déclarer une **classe template**, utilisez la syntaxe suivante :

```
template <typename T>
class MaClasse {
  // Déclaration des membres et méthodes
};
```

Les classes templates sont utilisées pour créer des classes génériques qui peuvent fonctionner avec différents types de données sans avoir à redéfinir la classe pour chaque type.

Syntaxe - Implémentation

Implémentez les méthodes d'une **classe template** comme suit :

```
template <typename T>
void MaClasse<T>::maMethode() {
  // Code de la méthode
}
```

Les classes templates permettent de créer des classes qui peuvent être utilisées avec différents types de données.

Exemples d'utilisation - Classes génériques

Voici un exemple d'une classe **template** `Pile` pouvant stocker différents types de données :

```
template <typename T>
class Pile {
public:
  void empiler(const T &element);
  T depiler();
  // ...
};
```

Les classes génériques permettent d'écrire des classes réutilisables pour différents types de données.

Exemples d'utilisation - Conteneurs réutilisables

```
Pile<int> pileEntiers;
pileEntiers.empiler(42);

Pile<std::string> pileStrings;
pileStrings.empiler("Hello, world!");
```

Les conteneurs réutilisables illustrés ici sont des **Piles**. Ils permettent de stocker des données de différents types, comme des entiers et des chaînes de caractères.

Héritage et classes templates

Hériter d'une classe template

Une classe template peut hériter d'une autre classe **template** ou d'une classe **non-template**.

```
template<typename T>
class Base {
  // ...
};

template<typename T>
class Derived : public Base<T> {
  // ...
};
```

```
// Usage
Derived<int> derivedInt;
```

Syntaxe

```
template <typename T>
class Derived: public Base<T> {
  // ...
};
```

La syntaxe de cet exemple montre comment hériter d'une classe de base avec un paramètre de modèle dans C++. Utiliser "template" pour déclarer un modèle et "typename" pour définir un type générique.

Exemples d'utilisation

Voici un exemple d'**héritage** entre deux **classes templates** :

```
template <typename T>
class Base {
  // ...
};

template <typename T>
class Derived: public Base<T> {
  // ...
};
```

Les classes templates permettent de créer des classes génériques qui peuvent être instanciées avec différents types de données. L'héritage fonctionne de manière similaire à celui des classes non-templates.

Spécialisation de templates - Classe

Pourquoi spécialiser ?

La spécialisation de templates de classe permet d'adapter le comportement d'une **classe template** pour certains types spécifiques.

Exemple :

```
// Classe template générique
template <typename T>
class MyClass {
public:
  T func(T arg) {
    // Implémentation générale
  }
};

// Spécialisation pour le type int
template <>
class MyClass<int> {
public:
  int func(int arg) {
    // Implémentation spécifique pour int
  }
};
```

La spécialisation est souvent utilisée pour optimiser le comportement d'une classe template pour certains types particuliers ou pour gérer des cas spéciaux.

Syntaxe

```
template <>
class MaClasse<int> {
  // Déclaration des membres et méthodes spécifiques pour le type int
};
```

Cet exemple montre la spécialisation de templates en C++ pour le type `int`. La spécialisation de templates permet de définir une implémentation spécifique pour un type particulier.

Exemples d'utilisation

Voici un exemple de **spécialisation** d'une classe template `Pile` pour le type `bool` :

```
template <>
class Pile<bool> {
public:
  void empiler(bool element);
  bool depiler();
  // ...
};
```

L'utilisation de la **spécialisation** de templates permet d'adapter le comportement d'une classe template pour certains types spécifiques, améliorant ainsi la modularité et la réutilisabilité du code.