Contents

Préparation du TP5

Points clés à retenir sur les Templates

Lire ou relire le cours de Sébastien Fourey, modèle (templates) chapître 3 pages 93 à 102

Définition des Templates :

Notion de généricité.
Les templates permettent de définir des classes et des fonctions génériques qui peuvent être utilisées avec différents types de données.

Syntaxe de base pour une classe template :

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

Syntaxe de base pour une fonction template :

```
template <typename T>
void myFunction(T param) {
        // Définition de la fonction
}
```

Utilisation des Templates :

Les templates permettent de réduire la duplication de code en fournissant des implémentations génériques.

Exemple d'utilisation d'une classe template :

```
MyClass<int> intClass;
MyClass<double> doubleClass;
```

Exemple d'utilisation d'une fonction template :

```
myFunction(10);
myFunction(3.14);
```

Spécialisation des Templates :

Il est possible de spécialiser les templates pour fournir des implémentations spécifiques pour certains types de données.

L'exemple (dans l'énoncé du TP5) des ensembles d'éléments de type `double` sera alors une spécialiation de votre travail.

Exemple de spécialisation d'une classe template :

```
template <>
class MyClass<bool> {
    // Implémentation spécifique pour bool
};
```

Exemple de spécialisation d'une fonction template :

```
template <>
void myFunction<bool>(bool param) {
    // Implémentation spécifique pour bool
}
```

Templates Variadiques :

Les templates variadiques permettent de définir des fonctions et des classes qui acceptent un nombre variable de types de données.

Exemple de fonction template variadique :

```
template <typename... Args>
void myVariadicFunction(Args... args) {
    // Implémentation de la fonction
}
```

Erreurs à éviter avec les Templates

Erreurs de Typage :

Assurez-vous que les types utilisés avec les templates sont compatibles avec les opérations effectuées dans les implémentations.

Exemple d'erreur de typage :

```
template <typename T>
void myFunction(T param) {
   param++; // Erreur si T n'est pas un type compatible avec l'opérateur ++
}
```

Erreurs de Compilation :

Les erreurs de compilation liées aux templates peuvent être difficiles à diagnostiquer. Assurez-vous que les templates sont correctement définis et utilisés.

Exemple d'erreur de compilation :

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

MyClass<int> intClass; // OK
MyClass<void> voidClass; // Erreur de compilation
```

Erreurs de Spécialisation :

Assurez-vous que les spécialisations de templates sont correctement définies et ne causent pas de conflits avec les définitions génériques.

Exemple d'erreur de spécialisation :

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

template <>
class MyClass<int> {
    int value;
};

template <>
class MyClass<int> {
    double value; // Erreur de redéfinition
};
```

Erreurs de Surcharge :

Les fonctions templates peuvent être surchargées, mais il est important de s'assurer que les surcharges sont correctement définies et ne causent pas de conflits.

Exemple d'erreur de surcharge :

```
template <typename T>
void myFunction(T param) {
    // Implémentation générique
}

void myFunction(int param) {
    // Surcharge spécifique pour int
}

myFunction(3.14); // Appelle la version générique
myFunction(3); // Appelle la version spécifique pour int
```

Erreurs de Conversion :

Assurez-vous que les conversions de types sont correctement gérées dans les templates.

Exemple d'erreur de conversion :

```
template <typename T>
void myFunction(T param) {
    double result = param; // Erreur si T n'est pas convertible en double
    }
```

Erreurs de Dépendance :

Les dépendances entre les templates doivent être correctement gérées pour éviter les erreurs de compilation.

Exemple d'erreur de dépendance :

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

template <typename T>
```

```cpp
class DependentClass {
    MyClass<T> member;
};

DependentClass<int> dependentClass; // OK
DependentClass<void> dependentClass; // Erreur de compilation
```

Différentes écritures et usages des templates

Élément	Description
Classe Template	Définit une classe générique qui peut être utilisée avec différents types de données.
Fonction Template	Définit une fonction générique qui peut être utilisée avec différents types de données.
Méthode Template dans une Classe Non-Template	Définit une méthode générique dans une classe non-template.
Méthode Template dans une Classe Template	Définit une méthode générique dans une classe template.
Spécialisation de Classe Template	Fournit une implémentation spécifique pour un type de donnée particulier.
Spécialisation Partielle de Classe Template	Fournit une implémentation spécifique pour un sous-ensemble de types de données.
Spécialisation de Fonction Template	Fournit une implémentation spécifique pour un type de donnée particulier.
Template Variadique	Définit une fonction qui accepte un nombre variable de types de données.
Template avec Paramètres par Défaut	Définit une classe template avec un paramètre par défaut.
Template avec Paramètres Non-Type	Définit une classe template avec un paramètre non-type.
Template avec Paramètres Template	Définit une classe template qui accepte un autre template comme paramètre.
Template avec Paramètres de Type et Non-Type	Définit une classe template avec des paramètres de type et non-type.
Template avec Paramètres de Type et Template	Définit une classe template avec des paramètres de type et template.
Template avec Paramètres de Type, Non-Type et Template	Définit une classe template avec des paramètres de type, non-type et template.
Template avec Paramètres de Type, Non-Type, Template et Par Défaut	Définit une classe template avec des paramètres de type, non-type, template et par défaut.
Template avec Paramètres de Type, Non-Type, Template, Par Défaut et Variadique	Définit une classe template avec des paramètres de type, non-type, template, par défaut et variadique.
Template avec Paramètres de Type, Non-Type, Template, Par Défaut, Variadique et Spécialisation	Définit une classe template avec des paramètres de type, non-type, template, par défaut, variadique et spécialisation.

Exemples

Classe Template

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

Fonction Template

```cpp
template <typename T>
void myFunction(T param) {
    // Définition de la fonction
}
```

Méthode Template dans une Classe Non-Template

```
class MyClass {
public:
    template <typename T>
    void myMethod(T param) {
        // Définition de la méthode
    }
};
```

Méthode Template dans une Classe Template

```
template <typename T>
class MyClass {
public:
    template <typename U>
    void myMethod(U param) {
        // Définition de la méthode
    }
};
```

Spécialisation de Classe Template

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

template <>
class MyClass<bool> {
    // Définition de la classe spécialisée pour bool
};
```

Spécialisation Partielle de Classe Template

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

template <typename T>
class MyClass<T*> {
    // Définition de la classe spécialisée pour les pointeurs
};
```

Spécialisation de Fonction Template

```
template <typename T>
void myFunction(T param) {
    // Définition de la fonction générique
}

template <>
void myFunction<bool>(bool param) {
    // Définition de la fonction spécialisée pour bool
}
```

Template Variadique

```
template <typename... Args>
void myVariadicFunction(Args... args) {
```

```
    // Définition de la fonction variadique
}
```

Template avec Paramètres par Défaut

```
template <typename T = int>
class MyClass {
    // Définition de la classe avec paramètre par défaut
};
```

Template avec Paramètres Non-Type

```
template <int N>
class MyClass {
    // Définition de la classe avec paramètre non-type
};
```

Template avec Paramètres Template

```
template <template <typename> class Container>
class MyClass {
    // Définition de la classe avec paramètre template
};
```

Template avec Paramètres de Type et Non-Type

```
template <typename T, int N>
class MyClass {
    // Définition de la classe avec paramètres de type et non-type
};
```

Template avec Paramètres de Type et Template

```
template <typename T, template <typename> class Container>
class MyClass {
    // Définition de la classe avec paramètres de type et template
};
```

Template avec Paramètres de Type, Non-Type et Template

```
template <typename T, int N, template <typename> class Container>
class MyClass {
    // Définition de la classe avec paramètres de type, non-type et template
};
```

Template avec Paramètres de Type, Non-Type, Template et Par Défaut

```
template <typename T = int, int N = 10, template <typename> class Container = std::vector>
class MyClass {
    // Définition de la classe avec paramètres de type, non-type, template et par défaut
};
```

Template avec Paramètres de Type, Non-Type, Template, Par Défaut et Variadique

```
template <typename T = int, int N = 10, template <typename> class Container = std::vector, typename... Args>
class MyClass {
    // Définition de la classe avec paramètres de type, non-type, template, par défaut et variadique
};
```

Template avec Paramètres de Type, Non-Type, Template, Par Défaut, Variadique et Spécialisation **Remarque:** `std::vector` *sera vu dans le prochain TP sur le STL Chapitre 5 pages 111 à 122*

```
template <typename T = int, int N = 10, template <typename> class Container = std::vector, typename... Args>
class MyClass {
    // Définition de la classe avec paramètres de type, non-type, template, par défaut et variadique
};

template <>
class MyClass<bool, 5, std::vector, int, double> {
    // Définition de la classe spécialisée
};
```

Interactions des templates avec les notions de POO

Notion de POO	Interaction avec les Templates
Encapsulation	Les templates peuvent être utilisés pour encapsuler des données et des comportements génériques.
Héritage	Les classes templates peuvent hériter d'autres classes templates ou de classes non-templates.
Polymorphisme	Les templates permettent un polymorphisme statique (résolu à la compilation) via les fonctions et classes génériques.
Surcharge	Les fonctions templates peuvent être surchargées pour fournir des implémentations spécifiques pour certains types de données.
Spécialisation	Les templates peuvent être spécialisés pour fournir des implémentations spécifiques pour certains types de données.
Héritage et Templates	Les classes templates peuvent être utilisées dans des hiérarchies d'héritage pour créer des classes dérivées génériques.
Polymorphisme et Templates	Les templates permettent de définir des fonctions et des classes génériques qui peuvent être utilisées avec différents types de données.
Encapsulation et Templates	Les templates peuvent être utilisés pour encapsuler des données et des comportements génériques dans des classes.
Surcharge et Templates	Les fonctions templates peuvent être surchargées pour fournir des implémentations spécifiques pour certains types de données.
Spécialisation et Templates	Les templates peuvent être spécialisés pour fournir des implémentations spécifiques pour certains types de données.
Héritage et Spécialisation	Les classes templates dérivées peuvent être spécialisées pour fournir des implémentations spécifiques pour certains types de données.
Polymorphisme et Spécialisation	Les fonctions templates spécialisées peuvent être utilisées pour fournir des implémentations spécifiques pour certains types de données.
Encapsulation et Spécialisation	Les classes templates spécialisées peuvent être utilisées pour encapsuler des données et des comportements spécifiques pour certains types de données.
Surcharge et Spécialisation	Les fonctions templates spécialisées peuvent être surchargées pour fournir des implémentations spécifiques pour certains types de données.
Héritage et Encapsulation	Les classes templates dérivées peuvent être utilisées pour encapsuler des données et des comportements génériques.
Polymorphisme et Encapsulation	Les classes templates peuvent être utilisées pour encapsuler des données et des comportements génériques tout en permettant le polymorphisme.
Surcharge et Encapsulation	Les fonctions templates peuvent être utilisées pour encapsuler des comportements génériques tout en permettant la surcharge.
Spécialisation et Encapsulation	Les classes templates spécialisées peuvent être utilisées pour encapsuler des données et des comportements spécifiques pour certains types de données.
Héritage et Polymorphisme	Les classes templates dérivées peuvent être utilisées pour permettre le polymorphisme tout en héritant de classes génériques.
Polymorphisme et Surcharge	Les fonctions templates peuvent être utilisées pour permettre le polymorphisme tout en permettant la surcharge.

Notion de POO	Interaction avec les Templates
Encapsulation et Polymorphisme	Les classes templates peuvent être utilisées pour encapsuler des données et des comportements génériques tout en permettant le polymorphisme.
Surcharge et Polymorphisme	Les fonctions templates peuvent être utilisées pour permettre le polymorphisme tout en permettant la surcharge.
Spécialisation et Polymorphisme	Les classes templates spécialisées peuvent être utilisées pour permettre le polymorphisme tout en fournissant des implémentations spécifiques pour certains types de données.
Héritage et Surcharge	Les classes templates dérivées peuvent être utilisées pour permettre la surcharge tout en héritant de classes génériques.
Polymorphisme et Surcharge	Les fonctions templates peuvent être utilisées pour permettre le polymorphisme tout en permettant la surcharge.
Encapsulation et Surcharge	Les classes templates peuvent être utilisées pour encapsuler des données et des comportements génériques tout en permettant la surcharge.
Spécialisation et Surcharge	Les classes templates spécialisées peuvent être utilisées pour encapsuler des données et des comportements spécifiques pour certains types de données tout en permettant la surcharge.

Exemples

Encapsulation

```
template <typename T>
class MyClass {
private:
    T data;
public:
    void setData(T value) { data = value; }
    T getData() const { return data; }
};
```

Héritage

```
template <typename T>
class BaseClass {
public:
    virtual void myMethod() = 0;
};

template <typename T>
class DerivedClass : public BaseClass<T> {
public:
    void myMethod() override {
        // Implémentation de la méthode
    }
};
```

Polymorphisme

```
template <typename T>
void myFunction(T param) {
    // Implémentation générique
}
```

Surcharge

```
template <typename T>
void myFunction(T param) {
    // Implémentation générique
```

```
}

void myFunction(int param) {
    // Implémentation spécifique pour int
}
```

Spécialisation

```
template <typename T>
class MyClass {
    // Implémentation générique
};

template <>
class MyClass<bool> {
    // Implémentation spécifique pour bool
};
```

Héritage et Templates

```
template <typename T>
class BaseClass {
    // Implémentation de la classe de base
};

template <typename T>
class DerivedClass : public BaseClass<T> {
    // Implémentation de la classe dérivée
};
```

Polymorphisme et Templates

```
template <typename T>
void myFunction(T param) {
    // Implémentation générique
}
```

Encapsulation et Templates

```
template <typename T>
class MyClass {
private:
    T data;
public:
    void setData(T value) { data = value; }
    T getData() const { return data; }
};
```

Surcharge et Templates

```
template <typename T>
void myFunction(T param) {
    // Implémentation générique
}

void myFunction(int param) {
    // Implémentation spécifique pour int
}
```

Spécialisation et Templates

```
template <typename T>
class MyClass {
    // Implémentation générique
};

template <>
class MyClass<bool> {
    // Implémentation spécifique pour bool
};
```

Héritage et Spécialisation

```
template <typename T>
class BaseClass {
    // Implémentation de la classe de base
};

template <typename T>
class DerivedClass : public BaseClass<T> {
    // Implémentation de la classe dérivée
};

template <>
class DerivedClass<bool> : public BaseClass<bool> {
    // Implémentation spécifique pour bool
};
```

Polymorphisme et Spécialisation

```
template <typename T>
void myFunction(T param) {
    // Implémentation générique
}

template <>
void myFunction<bool>(bool param) {
    // Implémentation spécifique pour bool
}
```

Encapsulation et Spécialisation

```
template <typename T>
class MyClass {
    // Implémentation générique
};

template <>
class MyClass<bool> {
private:
    bool data;
public:
    void setData(bool value) { data = value; }
    bool getData() const { return data; }
};
```

Surcharge et Spécialisation

```cpp
template <typename T>
void myFunction(T param) {
    // Implémentation générique
}

template <>
void myFunction<bool>(bool param) {
    // Implémentation spécifique pour bool
}

void myFunction(int param) {
    // Implémentation spécifique pour int
}
```

Héritage et Encapsulation

```cpp
template <typename T>
class BaseClass {
    // Implémentation de la classe de base
};

template <typename T>
class DerivedClass : public BaseClass<T> {
private:
    T data;
public:
    void setData(T value) { data = value; }
    T getData() const { return data; }
};
```

Polymorphisme et Encapsulation

```cpp
template <typename T>
class MyClass {
private:
    T data;
public:
    virtual void setData(T value) { data = value; }
    virtual T getData() const { return data; }
};
```

Surcharge et Encapsulation

```cpp
template <typename T>
void myFunction(T param) {
    // Implémentation générique
}

void myFunction(int param) {
    // Implémentation spécifique pour int
}
```

Spécialisation et Encapsulation

```cpp
template <typename T>
class MyClass {
    // Implémentation générique
};

template <>
```

```
class MyClass<bool> {
private:
    bool data;
public:
    void setData(bool value) { data = value; }
    bool getData() const { return data; }
};
```

Héritage et Polymorphisme

```
template <typename T>
class BaseClass {
public:
    virtual void myMethod() = 0;
};

template <typename T>
class DerivedClass : public BaseClass<T> {
public:
    void myMethod() override {
        // Implémentation de la méthode
    }
};
```

Polymorphisme et Surcharge

```
template <typename T>
void myFunction(T param) {
    // Implémentation générique
}

void myFunction(int param) {
    // Implémentation spécifique pour int
}
```

Encapsulation et Polymorphisme

```
template <typename T>
class MyClass {
private:
    T data;
public:
    virtual void setData(T value) { data = value; }
    virtual T getData() const { return data; }
};
```

Surcharge et Polymorphisme

```
template <typename T>
void myFunction(T param) {
    // Implémentation générique
}

void myFunction(int param) {
    // Implémentation spécifique pour int
}
```

Spécialisation et Polymorphisme

```
template <typename T>
class MyClass {
    // Implémentation générique
};

template <>
class MyClass<bool> {
private:
    bool data;
public:
    virtual void setData(bool value) { data = value; }
    virtual bool getData() const { return data; }
};
```

Héritage et Surcharge

```
template <typename T>
class BaseClass {
public:
    virtual void myMethod() = 0;
};

template <typename T>
class DerivedClass : public BaseClass<T> {
public:
    void myMethod() override {
        // Implémentation de la méthode
    }
};
```

Polymorphisme et Surcharge

```
template <typename T>
void myFunction(T param) {
    // Implémentation générique
}

void myFunction(int param) {
    // Implémentation spécifique pour int
}
```

Encapsulation et Surcharge

```
template <typename T>
class MyClass {
private:
    T data;
public:
    void setData(T value) { data = value; }
    T getData() const { return data; }
};
```

Spécialisation et Surcharge

```
template <typename T>
class MyClass {
    // Implémentation générique
};

template <>
```

```
class MyClass<bool> {
private:
    bool data;
public:
    void setData(bool value) { data = value; }
    bool getData() const { return data; }
};
```

Méthode incrémentale du TP

But du sujet, vous devrez créer à partir d'un exemple donné dans l'énoncé (les ensembles de type `double`), des classes, des méthodes, des fonctions génériques pour des ensembles de n'importe quel type (classiques, pointeurs, propre structure ou classe ...).

Analyse de la classe `SetOfDouble` pour préparer `Set`

Élément	`SetOfDouble`
Données Membres	
Classe Interne `Node`	
- Constructeur	`Node(double value, Node * next = nullptr)`
- Destructeur	`~Node()`
- Méthode `getValue`	`double getValue() const`
- Méthode `getNext`	`Node * getNext() const`
- Méthode `setNext`	`void setNext(Node * next)`
- Données Membres	`double value`
Pointeur vers le Premier Nœud	`Node * list`
Méthodes	
Constructeurs	
- Constructeur par Défaut	`SetOfDouble()`
- Constructeur pour Créer un Singleton	`SetOfDouble(double x)`
Méthode pour Vider l'Ensemble	`void clear()`
Destructeur	`~SetOfDouble()`
Méthode pour Vérifier si l'Ensemble est Vide	`bool isEmpty() const`
Méthode pour Envoyer les Éléments à un Flux de Sortie	`std::ostream & flush(std::ostream & out) const`
Méthode pour Vérifier l'Appartenance d'un Élément	`bool contains(double x) const`
Méthode pour Vérifier l'Inclusion d'un Ensemble	`bool isSubsetOf(const SetOfDouble & other) const`
Méthode pour Insérer un Élément	`void insert(double x)`
Méthode pour Supprimer un Élément	`void remove(double x)`
Méthode pour Insérer Tous les Éléments dans un Autre Ensemble	`void insertInto(SetOfDouble & other) const`
Méthode pour Supprimer un Ensemble d'un Autre Ensemble	`void removeFrom(SetOfDouble & other) const`
Opérateur d'Affectation	`SetOfDouble & operator=(const SetOfDouble & other)`
Constructeur de Copie	`SetOfDouble(const SetOfDouble & other)`
Méthode Statique pour Supprimer la Première Occurrence d'une Valeur	`static Node * remove(Node * list, double x)`
Opérateurs et Fonctions amies	
Opérateur de Flux de Sortie	`std::ostream & operator<<(std::ostream & out, const SetOfDouble & s)`
Fonction pour Créer un Singleton	`SetOfDouble singleton(double x)`
Fonction pour Créer un Ensemble Vide	`SetOfDouble emptySet()`
Opérateur d'Égalité	`bool operator==(const SetOfDouble & a, const SetOfDouble & b)`
Opérateur de Comparaison pour Vérifier l'Inclusion	`bool operator<(const SetOfDouble & a, const SetOfDouble & b)`
Opérateur de Comparaison pour Vérifier la Contenance	`bool operator>(const SetOfDouble & a, const SetOfDouble & b)`

Élément	SetOfDouble
Opérateur d'Union	`SetOfDouble operator|(const SetOfDouble & a, const SetOfDouble & b)`
Opérateur de Différence	`SetOfDouble operator-(const SetOfDouble & a, const SetOfDouble & b)`
Opérateur de Différence Symétrique	`SetOfDouble operator^(const SetOfDouble & a, const SetOfDouble & b)`
Opérateur d'Intersection	`SetOfDouble operator&(const SetOfDouble & a, const SetOfDouble & b)`
Fonction pour Appliquer une Fonction à Chaque Élément de l'Ensemble	`SetOfDouble image(const SetOfDouble & set, double (*function)(double))`

Consigne particulière

Pourquoi définir les Modèles (Template) dans le Fichier d'en-tête ?

Réutilisabilité :

En définissant les modèles entièrement dans le fichier d'en-tête, vous permettez à d'autres fichiers d'inclure ce fichier d'en-tête et d'utiliser les modèles sans avoir besoin de lier séparément les fichiers d'implémentation.

Éviter les erreurs de liaison :

Les modèles en C++ sont instanciés à la compilation. Si les définitions des modèles sont séparées dans des fichiers d'implémentation (.cpp), le compilateur peut ne pas être capable de trouver les définitions nécessaires pour instancier les modèles, ce qui peut entraîner des erreurs de liaison.

Exemple Fichier MyTemplate.h

```
#ifndef MYTEMPLATE_H
#define MYTEMPLATE_H

#include <iostream>

// Déclaration de la classe template
template <typename T>
class MyClass {
private:
    T data;
public:
    MyClass(T value);
    void setData(T value);
    T getData() const;
    void display() const;
};

#endif // MYTEMPLATE_H
```

Fichier MyTemplate.cpp

```
#include "MyTemplate.h"

// Définition de la classe template
template <typename T>
MyClass<T>::MyClass(T value) : data(value) {}

template <typename T>
void MyClass<T>::setData(T value) {
    data = value;
```

```
}

template <typename T>
T MyClass<T>::getData() const {
    return data;
}

template <typename T>
void MyClass<T>::display() const {
    std::cout << "Data: " << data << std::endl;
}
```

Fichier main.cpp

```
#include "MyTemplate.h"

int main() {
    // Utilisation de la classe template
    MyClass<int> intObj(10);
    intObj.display();

    MyClass<double> doubleObj(3.14);
    doubleObj.display();

    return 0;
}
```

Problème Si vous compilez ce code, vous obtiendrez des erreurs de liaison. Le compilateur ne pourra pas trouver les définitions des méthodes de la classe template MyClass dans le fichier main.cpp parce que les définitions sont dans un fichier séparé (`MyTemplate.cpp`).

Solution Pour éviter ces erreurs de liaison, vous devez définir entièrement les modèles dans le fichier d'en-tête.

Voici comment vous pouvez le faire :

Fichier MyTemplate.h

```
#ifndef MYTEMPLATE_H
#define MYTEMPLATE_H

#include <iostream>

// Définition de la classe template
template <typename T>
class MyClass {
private:
    T data;
public:
    MyClass(T value) : data(value) {}

    void setData(T value) {
        data = value;
    }

    T getData() const {
        return data;
    }

    void display() const {
        std::cout << "Data: " << data << std::endl;
```

```
    }
};

#endif // MYTEMPLATE_H
```

Fichier main.cpp

```
#include "MyTemplate.h"

int main() {
    // Utilisation de la classe template
    MyClass<int> intObj(10);
    intObj.display();

    MyClass<double> doubleObj(3.14);
    doubleObj.display();

    return 0;
}
```

Simplicité :

En gardant les définitions des modèles dans le fichier d'en-tête, vous simplifiez la structure de votre projet et facilitez la maintenance du code.