



POO en C++ - Template -

Groupe étudiant : CIR2

Nils BEAUSSÉ

E-Mail : nils.beausse@isen-ouest.yncrea.fr

Les Templates (modèles)



Les Templates (modèles)



Qu'est-ce qu'un template ?



Qu'est-ce qu'un template ?

- Les **templates (modèles ou patrons)** servent à écrire un code **indépendant** du type des données avec lesquelles il sera finalement utilisé \Rightarrow C'est la **programmation générique**.



Qu'est-ce qu'un template ?

- Les **templates (modèles ou patrons)** servent à écrire un code **indépendant** du type des données avec lesquelles il sera finalement utilisé ⇒ C'est la **programmation générique**.
- Les **fonctions/méthodes templates** sont des fonctions qui utilisent les templates ⇒ elles peuvent être utilisée de la même façon avec différents types selon le bon vouloir de l'utilisateur.

Qu'est-ce qu'un template ?

- Les **templates (modèles ou patrons)** servent à écrire un code **indépendant** du type des données avec lesquelles il sera finalement utilisé ⇒ C'est la **programmation générique**.
- Les **fonctions/méthodes templates** sont des fonctions qui utilisent les templates ⇒ elles peuvent être utilisée de la même façon avec différents types selon le bon vouloir de l'utilisateur.
- Les **classes templates** sont des classes qui utilisent les templates ⇒ leurs attributs et méthodes peuvent être typés à l'utilisation.

Fonctions template

Pourquoi les Templates, cas pratique :

```
void swap(int& a, int& b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(float& a, float& b){  
    float temp = a;  
    a = b;  
    b = temp;  
}
```

Pourquoi les Templates, cas pratique :

- Soit les deux fonctions suivantes :

```
void swap(int& a, int& b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(float& a, float& b){  
    float temp = a;  
    a = b;  
    b = temp;  
}
```

Pourquoi les Templates, cas pratique :

- Soit les deux fonctions suivantes :

```
void swap(int& a, int& b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(float& a, float& b){  
    float temp = a;  
    a = b;  
    b = temp;  
}
```

- même logique avec **différents types**. Comment remédier à cela ?

Pourquoi les Templates, cas pratique :

- Soit les deux fonctions suivantes :

```
void swap(int& a, int& b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(float& a, float& b){  
    float temp = a;  
    a = b;  
    b = temp;  
}
```

- même logique avec **différents types**. **Comment remédier à cela ?**

→ **On pourrait les surcharger** mais ça obligerait quand même à écrire les deux codes !

Pourquoi les Templates, cas pratique :

- Soit les deux fonctions suivantes :

```
void swap(int& a, int& b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(float& a, float& b){  
    float temp = a;  
    a = b;  
    b = temp;  
}
```

- même logique avec **différents types**. **Comment remédier à cela ?**
 - **On pourrait les surcharger** mais ça obligerait quand même à écrire les deux codes !
 - Et si il n'y a que des opérations standard par exemple, comme c'est le cas ici -> on aimerait pouvoir swapper tous les objets possible et imaginable sans avoir à tout réécrire !

Pourquoi les Templates, cas pratique :

→ Et si il n'y a que des opérations standard par exemple, comme c'est le cas ici -> on aimerait pouvoir swapper tous les objets possible et imaginable sans avoir à tout réécrire !



Pourquoi les Templates, cas pratique :

→ Et si il n'y a que des opérations standard par exemple, comme c'est le cas ici -> on aimerait pouvoir swapper tous les objets possible et imaginable sans avoir à tout réécrire !

```
void swap(int&a, int&b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

Pourquoi les Templates, cas pratique :

→ Et si il n'y a que des opérations standard par exemple, comme c'est le cas ici -> on aimerait pouvoir swapper tous les objets possible et imaginable sans avoir à tout réécrire !

```
void swap(int&a, int&b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(float&a, float&b){  
    float temp = a;  
    a = b;  
    b = temp;  
}
```


Pourquoi les Templates, cas pratique :

→ Et si il n'y a que des opérations standard par exemple, comme c'est le cas ici -> on aimerait pouvoir swapper tous les objets possible et imaginable sans avoir à tout réécrire !

```
void swap(int&a, int&b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(float&a, float&b){  
    float temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(bool&a, bool&b){  
    bool temp = a;  
    a = b;  
    b = temp;  
}
```

Pourquoi les Templates, cas pratique :

→ Et si il n'y a que des opérations standard par exemple, comme c'est le cas ici -> on aimerait pouvoir swapper tous les objets possible et imaginable sans avoir à tout réécrire !

```
void swap(int&a, int&b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(float&a, float&b){  
    float temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(bool&a, bool&b){  
    bool temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(double&a, double &b){  
    double temp = a;  
    a = b;  
    b = temp;  
}
```

Pourquoi les Templates, cas pratique :

→ Et si il n'y a que des opérations standard par exemple, comme c'est le cas ici -> on aimerait pouvoir swapper tous les objets possible et imaginable sans avoir à tout réécrire !

```
void swap(int&a, int&b){
    int temp = a;
    a = b;
    b = temp;
}
```

```
void swap(float&a, float&b){
    float temp = a;
    a = b;
    b = temp;
}
```

```
void swap(bool&a, bool&b){
    bool temp = a;
    a = b;
    b = temp;
}
```

```
void swap(ma_classe&a, ma_classe &b){
    ma_classe temp = a;
    a = b;
    b = temp;
}
```

```
void swap(double&a, double &b){
    double temp = a;
    a = b;
    b = temp;
}
```

Pourquoi les Templates, cas pratique :

→ Et si il n'y a que des opérations standard par exemple, comme c'est le cas ici -> on aimerait pouvoir swapper tous les objets possible et imaginable sans avoir à tout réécrire !

```
void swap(int&a, int&b){
    int temp = a;
    a = b;
    b = temp;
}
```

```
void swap(float&a, float&b){
    float temp = a;
    a = b;
    b = temp;
}
```

```
void swap(ma_classe&a, ma_classe &b){
    ma_classe temp = a;
    a = b;
    b = temp;
}
```

```
void swap(ma_structure&a, ma_structure &b){
    ma_structure temp = a;
    a = b;
    b = temp;
}
```

```
void swap(bool&a, bool&b){
    bool temp = a;
    a = b;
    b = temp;
}
```

```
void swap(double&a, double &b){
    double temp = a;
    a = b;
    b = temp;
}
```

Un code fonctionnant avec tout type !

Même logique avec **différents types**. Comment factoriser le code ?

→ Utilisation des **fonctions templates** ⇒ **définition d'un type générique**

- Syntaxe :

```
template<typename T> return_type function_name(...){  
    ...  
}
```

Swap générique

- Dans le fichier swap.h :



Swap générique

- Dans le fichier swap.h :

```
template<typename T> void swap(T& a, T& b){  
    T temp = a;  
    a = b;  
    b = temp;  
}
```

Swap générique

- Dans le fichier swap.h :

```
template<typename T> void swap(T& a, T& b){  
    T temp = a;  
    a = b;  
    b = temp;  
}
```

- La fonction swap doit être **déclarée et définie** dans un fichier d'extension “.h” (ou .hpp) → Pourquoi ?

Swap générique

- Dans le fichier swap.h :

```
template<typename T> void swap(T& a, T& b){  
    T temp = a;  
    a = b;  
    b = temp;  
}
```

- La fonction swap doit être **déclarée et définie** dans un fichier d'extension “.h” (ou .hpp) → Pourquoi ?
- Les différentes variantes de la fonction swap vont être générées par le compilateur en se basant sur les types des arguments fournis lors de l'appel à la fonction

Swap générique

Le nom du template
= type générique

- Dans le fichier swap.h :

```
template<typename T> void swap(T& a, T& b){  
    T temp = a;  
    a = b;  
    b = temp;  
}
```

On peut déclarer plusieurs
types séparés par des
virgules

- La fonction swap doit être **déclarée et définie** dans un fichier d'extension “.h” (ou .hpp) → Pourquoi ?
- Les différentes variantes de la fonction swap vont être générées par le compilateur en se basant sur les types des arguments fournis lors de l'appel à la fonction

Swap générique

- Dans le fichier "swaptest.cpp" :



Swap générique

- Dans le fichier "swaptest.cpp" :

```
#include "swap.h"
#include <iostream>
using std::cout;
using std::endl;

int main(){
    float f1 = 3.5; float f2 = 1.1;
    swap(f1, f2);
    cout << f1 << " - " << f2 << endl;

    int i1 = 1; int i2 = 42;
    swap(i1, i2);
    cout << i1 << " - " << i2 << endl;

    return 0;
}
```

Swap générique

- Dans le fichier "swaptest.cpp" :

```
#include "swap.h"
#include <iostream>
using std::cout;
using std::endl;

int main(){
    float f1 = 3.5; float f2 = 1.1;
    swap(f1, f2);
    cout << f1 << " - " << f2 << endl;

    int i1 = 1; int i2 = 42;
    swap(i1, i2);
    cout << i1 << " - " << i2 << endl;

    return 0;
}
```

1.1 - 3.5
42 - 1

Swap générique

- Dans le fichier "swaptest.cpp" :

```
#include "swap.h"
#include <iostream>
using std::cout;
using std::endl;

int main(){
    float f1 = 3.5; float f2 = 1.1;
    swap(f1, f2);
    cout << f1 << " - " << f2 << endl;

    int i1 = 1; int i2 = 42;
    swap(i1, i2);
    cout << i1 << " - " << i2 << endl;

    return 0;
}
```

1.1 - 3.5
42 - 1

```
template<typename T> void swap(T& a, T& b){
    T temp = a;
    a = b;
    b = temp;
}
```

Swap générique

- Dans le fichier "swaptest.cpp" :

```
#include "swap.h"
#include <iostream>
using std::cout;
using std::endl;

int main(){
    float f1 = 3.5; float f2 = 1.1;
    swap(f1, f2);
    cout << f1 << " - " << f2 << endl;

    int i1 = 1; int i2 = 42;
    swap(i1, i2);
    cout << i1 << " - " << i2 << endl;

    return 0;
}
```

Génère automatiquement
swap<float>(float&, float&)

1.1 - 3.5
42 - 1

```
template<typename T> void swap(T& a, T& b){
    T temp = a;
    a = b;
    b = temp;
}
```

Swap générique

- Dans le fichier "swaptest.cpp" :

```
#include "swap.h"
#include <iostream>
using std::cout;
using std::endl;

int main(){
    float f1 = 3.5; float f2 = 1.1;
    swap(f1, f2);
    cout << f1 << " - " << f2 << endl;

    int i1 = 1; int i2 = 42;
    swap(i1, i2);
    cout << i1 << " - " << i2 << endl;

    return 0;
}
```

Génère automatiquement
swap<float>(float&, float&)

Génère automatiquement
swap<int>(int&, int&)

1.1 - 3.5
42 - 1

```
template<typename T> void swap(T& a, T& b){
    T temp = a;
    a = b;
    b = temp;
}
```


Swap générique

- Lors de la surcharge des fonctions, les fonctions non-templates sont appelées en premier
- Pour forcer l'appel de la fonction template, on utilise `<>`
- **Exemple** : Soit un fichier "f.h" (à gauche) et un fichier "ftest.cpp" (à droite)

```
//... all required lines
```

```
template<typename T> void f(T t){  
    cout << "f<T>(T)" << endl;  
}
```

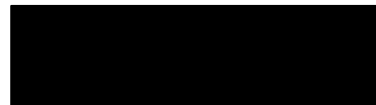
generic version

```
void f(int i){  
    cout << "f(int)" << endl;  
}
```

specialized version

```
#include "f.h"  
// ... all required lines
```

```
int main()  
{  
    → f(1)  
    return 0;  
}
```



Swap générique

- Lors de la surcharge des fonctions, les fonctions non-templates sont appelées en premier
- Pour forcer l'appel de la fonction template, on utilise `<>`
- Exemple : Soit un fichier "f.h" (à gauche) et un fichier "ftest.cpp" (à droite)

```
//... all required lines
```

```
template<typename T> void f(T t){  
    cout << "f<T>(T)" << endl;  
}
```

generic version

```
void f(int i){  
    cout << "f(int)" << endl;  
}
```

specialized version

```
#include "f.h"  
// ... all required lines
```

```
int main()  
{  
    → f(1)  
    return 0;  
}
```

f(int)

Swap générique

- Lors de la surcharge des fonctions, les fonctions non-templates sont appelées en premier
- Pour forcer l'appel de la fonction template, on utilise `<>`
- Exemple : Soit un fichier "f.h" (à gauche) et un fichier "ftest.cpp" (à droite)

```
//... all required lines
```

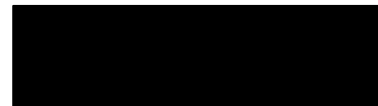
```
template<typename T> void f(T t){  
    cout << "f<T>(T)" << endl;  
}
```

generic version

```
void f(int i){  
    cout << "f(int)" << endl;  
}
```

specialized version

```
#include "f.h"  
// ... all required lines  
  
int main()  
{  
    → f(1.0)  
    return 0;  
}
```



Swap générique

- Lors de la surcharge des fonctions, les fonctions non-templates sont appelées en premier
- Pour forcer l'appel de la fonction template, on utilise `<>`
- Exemple : Soit un fichier "f.h" (à gauche) et un fichier "ftest.cpp" (à droite)

```
//... all required lines
```

```
template<typename T> void f(T t){  
    cout << "f<T>(T)" << endl;  
}
```

generic version

```
void f(int i){  
    cout << "f(int)" << endl;  
}
```

specialized version

```
#include "f.h"  
// ... all required lines  
  
int main()  
{  
    → f(1.0)  
    return 0;  
}
```

`f<T>(T)`

Swap générique

- Lors de la surcharge des fonctions, les fonctions non-templates sont appelées en premier
- Pour forcer l'appel de la fonction template, on utilise `<>`
- Exemple : Soit un fichier "f.h" (à gauche) et un fichier "ftest.cpp" (à droite)

```
//... all required lines
```

```
template<typename T> void f(T t){  
    cout << "f<T>(T)" << endl;  
}
```

generic version

```
void f(int i){  
    cout << "f(int)" << endl;  
}
```

specialized version

```
#include "f.h"  
// ... all required lines  
  
int main()  
{  
    → f<int>(1)  
    return 0;  
}
```



Swap générique

- Lors de la surcharge des fonctions, les fonctions non-templates sont appelées en premier
- Pour forcer l'appel de la fonction template, on utilise `<>`
- Exemple : Soit un fichier "f.h" (à gauche) et un fichier "ftest.cpp" (à droite)

```
//... all required lines
```

```
template<typename T> void f(T t){  
    cout << "f<T>(T)" << endl;  
}
```

generic version

```
void f(int i){  
    cout << "f(int)" << endl;  
}
```

specialized version

```
#include "f.h"  
// ... all required lines  
  
int main()  
{  
    → f<int>(1)  
    return 0;  
}
```

f<T>(T)

Swap générique

- Lors de la surcharge des fonctions, les fonctions non-templates sont appelées en premier
- Pour forcer l'appel de la fonction template, on utilise `<>`
- Exemple : Soit un fichier "f.h" (à gauche) et un fichier "ftest.cpp" (à droite)

```
//... all required lines
```

```
template<typename T> void f(T t){  
    cout << "f<T>(T)" << endl;  
}
```

generic version

```
void f(int i){  
    cout << "f(int)" << endl;  
}
```

specialized version

```
#include "f.h"  
// ... all required lines  
  
int main()  
{  
    → f<int>(1)  
    return 0;  
}
```


`f<T>(T)`

Parfois on doit aussi préciser le types explicitement si il y a possibilité de confusion pour le compilateur (par exemple entre un float et un double, etc.)

Multiples types génériques

On peut tout à fait définir plusieurs types template pour une même fonction :

```
template<typename T, typename U, typename V=int> void addition(T& resultat, U& b, V& c){
    T resul_inter = b*c;
    U temp = b*b;
    V temp = c*c;
    resultat = resul_inter + c;
}
```



Et même définir des **arguments par défauts** !! Ainsi :

addition<int, float> reviendra à **addition <int,float,int>** ici

(même règle que pour arg par défaut ds fonction -> les derniers uniquement par défauts (ou tout))

Si tout les arguments ont un type par défaut, il est autorisé, **depuis C++17** d'appeler la fonction tout à fait normalement : `addition(a,b,c)` sans `<>`. Avant C++17 il faudrait écrire `addition<>>(a,b,c)`

Multiple types génériques

Si **tout les arguments** ont un type par défaut, il est autorisé, **depuis C++17** d'appeler la fonction tout à fait normalement : `addition(a,b,c)` sans `<>`. Avant C++17 il faudrait écrire `addition<>(a,b,c)`

TOUJOURS OK !

```
template <typename T = int> T pif(){  
    return 2;  
};
```

```
int main() {  
    int a;  
    a = pif<int>();  
}
```

```
template <typename T = int> T pif(){  
    return 2;  
};
```

```
int main() {  
    float b;  
    b = pif<float>();  
}
```

OK si version C++ < C++17

```
template <typename T = int> T pif{  
    return 2;  
};
```

```
int main() {  
    int a;  
    a = pif<>();  
}
```

OK si version >= C++17 !

```
template <typename T = int> T pif{  
    return 2;  
};
```

```
int main() {  
    int a;  
    a = pif();  
}
```

Classe template

Déclaration d'une classe Template

- Syntaxe :

```
template<class T> class ClassName {  
    // attributs utilisant T  
    T attr1;  
    // méthodes utilisant T  
    return_type method1(...);  
};  
  
template<class T> return_type ClassName<T>::method1(...){  
    ...  
}
```

Méthode
template

Le nom de la
classe

Déclaration d'une classe Template

- Syntaxe :

En pratique dans les versions récentes de C++ il n'y a plus de différence entre `template <typename T>` et `template <class T>` en dehors de cas très très particulier

```
template<class T> class ClassName {  
    // attributs utilisant T  
    T attr1;  
    // méthodes utilisant T  
    return_type method1(...);  
};  
  
template<class T> return_type ClassName<T>::method1(...){  
    ...  
}
```

Méthode qui utilise T -> on reprecise le template devant

Le nom de la classe (on doit bien préciser T) sinon le compilateur pourrait penser qu'il s'agit d'une méthode template d'une classe normale !

Déclaration d'une classe Template

- Exemple ("Fraction.h")

```
template<class T>
class Fraction {
private:
    T num;    T den;
public:
    Fraction(T n, T d);
    Fraction<T> productFractions(Fraction<T> f);
    void print();
};

template<class T> Fraction<T>::Fraction(T n, T d) : num(n), den(d) {}

template<class T> Fraction<T> Fraction<T>::productFractions(Fraction<T> f) {
    return Fraction(f.num*num, f.den*den);
}

template<class T> void Fraction<T>::print() {
    std::cout << num << "/" << den;
}
```

Déclaration d'une classe Template

- Exemple ("Fraction.h") :

```
template<class T> class Fraction {  
private:  
    T num;    T den;  
public:  
    Fraction(T n, T d);  
    Fraction<T> productFractions(Fraction<T> f);  
    void print();  
};
```

```
template<class T> Fraction<T>::Fraction(T n, T d) : num(n), den(d) {}
```

```
template<class T> Fraction<T> Fraction<T>::productFractions(Fraction<T> f) {  
    return Fraction(f.num*num, f.den*den);  
}
```

```
template<class T> void Fraction<T>::print() {  
    std::cout << num << "/" << den;  
}
```

```
// main.cpp  
int main() {  
    Fraction<int> f0(2.0,8.0);  
    f0.print();  
    Fraction<double> f1(5.0,3.0);  
    Fraction<double> f2(2.0,4.0);  
    Fraction<double> f3 = f1.productFractions(f2);  
    f3.print();  
    return 0;  
}
```

Déclaration d'une classe Template

- Exemple ("Fraction.h") :

```
class Fraction {  
private:  
    int num;    int den;  
public:  
    Fraction(int n, int d);  
    int productFractions(Fraction f);  
    void print();  
};
```

```
template<typename T> Fraction::Fraction(T n, T d) : num((int)n), den((int)d)  
{}
```

```
Fraction Fraction::productFractions(Fraction f) {  
    return Fraction(f.num*num, f.den*den);  
}
```

```
void Fraction::print() {  
    std::cout << num << "/" << den;  
}
```

```
// main.cpp  
int main() {  
    Fraction f0(2.0,8.0);  
    f0.print();  
    Fraction f1(5.0,3.0);  
    Fraction f2(2.0,4.0);  
    Fraction f3 = f1.productFractions(f2);  
    f3.print();  
    return 0;  
}
```

Une classe normale
avec une méthode
template c'est aussi
possible !

Déclaration d'une classe Template

- Exemple ("Fraction.h") :

```
template<class T> class Fraction {  
private:  
    T num;    T den;  
public:  
    Fraction(T n, T d);  
    Fraction<T> productFractions(Fraction<T> f);  
    void print();  
};
```

```
template<class T> template<class T2> Fraction<T>::Fraction(T n, T d, T2 u) : num(n*u), den(d*u) {}
```

```
template<class T> Fraction<T> Fraction<T>::productFractions(Fraction<T> f) {  
    return Fraction(f.num*num, f.den*den);  
}
```

```
template<class T> void Fraction<T>::print() {  
    std::cout << num << "/" << den;  
}
```

```
// main.cpp  
int main() {  
    Fraction<int> f0(2.0,8.0,5);  
    f0.print();  
    Fraction<double> f1(5.0,3.0,2);  
    Fraction<double> f2(2.0,4.0,3);  
    Fraction<double> f3 = f1.productFractions(f2);  
    f3.print();  
    return 0;  
}
```

Une classe template
avec une méthode qui a
des template en plus
c'est aussi possible !

Template template ?

- On peut tout à fait utiliser des templates templates si on a une classe template qui doit utiliser une autre classe template par exemple.

```
template <class T> class Tableau
{
    // Définition de la classe template Tableau.
};

template <class U, class V, template <class T> class C=Tableau> class Dictionnaire
{
    C<U> Clef;
    C<V> Valeur;
};
```