

Cours 4

Templates & L/R-Value

C++ - Master 1

1. Fonctions-template
2. Classes-template
3. Spécialisations
4. Catégories de valeurs
5. Constructeur et opérateur de déplacement

Un template, ou patron, est un modèle qui sert à générer du code automatiquement.

On a des fonctions-template, qui permettent de créer des fonctions, et des classes-templates, qui permettent de créer des types.

Les templates permettent de faire du polymorphisme et de la généricité en C++.

Quelques exemples de templates que vous avez déjà rencontrés :

- les classes-template `std::vector`, `std::map`, et autres conteneurs
- les fonctions-template `std::move`, `std::make_unique` ou `std::min`

```
template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

mot-clef pour indiquer
qu'on crée un template

liste de paramètres
du template

```
template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << " " << std::endl;
}
```

nom du template

“Instancier un template” signifie que le compilateur va générer une instance du modèle.

Par exemple, `std::min<int>` **et** `std::min<std::string>` sont deux instances différentes du template `std::min`.

Pour instancier une fonction-template, on peut simplement appeler une instance de la fonction-template. Le compilateur va automatiquement instancier la fonction depuis le modèle lorsqu’il verra la ligne correspondante dans le code.

Attention ! Il faut que le compilateur ait vu le template pour pouvoir en faire une instanciation.

```
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    print_between_parentheses<std::string>("pouet");
}
```

le compilateur enregistre le template
dans sa base de données

```
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    print_between_parentheses<std::string>("pouet");
}
```


le compilateur va générer la fonction
`print_between_parentheses<std::string>`
à partir du modèle = instantiation

```
#include <iostream>
```

```
template <typename T>  
void print_between_parentheses(const T& value)  
{  
    std::cout << '(' << value << ')' << std::endl;  
}
```



```
int main()  
{  
    print_between_parentheses<std::string>("pouet");  
}
```

```
#include <iostream>
```

```
template <typename T>
```

```
void print_between_parentheses(const T& value)
```

```
{
```

```
    std::cout << '(' << value << ')' << std::endl;
```

```
}
```

```
int main()
```

```
{
```

```
    print_between_parentheses<std::string>("pouet");
```

```
}
```

la fonction est ajoutée à l'unité de compilation courante
(.o) et sera correctement liée au programme

```
void print_between_parentheses<std::string>(const std::string& value)
```

```
{
```

```
    std::cout << '(' << value << ')' << std::endl;
```

```
}
```



Si les paramètres du template sont utilisés dans la signature de la fonction, alors ceux-ci peuvent être automatiquement déduits au moment de l'appel à la fonction.

```
template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

```
template <typename T>
T cast_integer(int i)
{
    return static_cast<T>(i);
}
```

Si les paramètres du template sont utilisés dans la signature de la fonction, alors ceux-ci peuvent être automatiquement déduits. T est utilisé dans la signature, la déduction pourra se faire.

```
template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

```
template <typename T>
T cast_integer(int i)
{
    return static_cast<T>(i);
}
```

T n'est pas utilisé dans la signature, la déduction ne peut pas se faire

```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```



Est-ce que je connais une
fonction min à 2 paramètres ?

```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```



NON !

```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```



Est-ce que je connais une
fonction-template min à 2 paramètres ?

```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```

template <typename **T**>
T min(T v1, T v2) { ... }



OUI !


```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```

COMPILO



template <typename
T min(T v1, T v2)

Est-ce que les types des arguments
me permettent de déduire les
paramètres du template ?

```
#include <algorithm>
#include <iostream>
```

```
int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```

int

int

```
template <typename T>
T min(T v1, T v2) { ... }
```

T = int

T = int

COMPILO



OUI !

```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```

```
int min<int>(int v1, int v2) { ... }
```



J'instancie `std::min<int>`

```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```



Est-ce que je connais une
fonction min à 2 paramètres ?

```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```



NON !

```
#include <algorithm>
#include <iostream>
```

```
int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```



Est-ce que je connais une
fonction-template min à 2 paramètres ?

```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```

template <typename **T**>
T min(T v1, T v2) { ... }



OUI !

```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```

COMPILO



template <typename
T min(T v1, T v2)

Est-ce que les types des arguments
me permettent de déduire les
paramètres du template ?


```
#include <algorithm>
#include <iostream>
```

```
int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```

double

int

```
template <typename T>
T min(T v1, T v2) { ... }
```

T = double

T = int

COMPILO



NON !

```
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```



J'insulte le développeur !!

```
<source>: In function 'int main()':  
<source>:23:13: error: no matching function for call to 'min(double, int)'  
 23 |     std::min(1.3, 4);  
    |     ~~~~~^~~~~~  
In file included from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/string:50,  
    from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/bits/locale_classes.h:40,  
    from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/bits/ios_base.h:41,  
    from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/ios:42,  
    from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/ostream:38,  
    from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/iostream:39,  
    from <source>:1:  
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/bits/stl_algobase.h:230:5: note: candidate:  
'template<class _Tp> constexpr const _Tp& std::min(const _Tp&, const _Tp&)'  
 230 |     min(const _Tp& __a, const _Tp& __b)  
    |     ^~~  
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/bits/stl_algobase.h:230:5: note: template  
argument deduction/substitution failed:  
<source>:23:13: note: deduced conflicting types for parameter 'const _Tp' ('double' and 'int')  
 23 |     std::min(1.3, 4);  
    |     ~~~~~^~~~~~
```



```
template <typename T>  
struct Fraction  
{  
    T dividende = {};  
    T diviseur = {};  
};
```

mot-clef pour indiquer
qu'on définit un template

```
template <typename T>
```

```
struct Fraction
```

```
{
```

nom du template

```
    idende
```

```
    iseur = {};
```

```
};
```

liste de paramètres
du template

`Fraction<int>` **et** `Fraction<double>` sont des instances du template `Fraction`.

`Fraction<int>` **et** `Fraction<double>` sont des types, mais `Fraction` n'est pas un type.

Comme pour les fonctions-templates, afin d'instancier une classe-template, on peut utiliser une instance du template. Attention, pour que cela fonctionne, il faut que le compilateur ait eu connaissance du template. Pensez donc bien à mettre tout le code de vos templates dans les **headers**.



```
template <typename T>
class Printer
{
public:
    void parentheses(const T& v) const { std::cout << '(' << v << ')'; }
    void quote(const T& v) const { std::cout << '"' << v << '"'; }
};

int main()
{
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```

le compilateur enregistre le template
dans sa base de données

```
template <typename T>
class Printer
{
public:
    void parentheses(const T& v) const { std::cout << '(' << v << ')'; }
    void quote(const T& v) const { std::cout << '"' << v << '"'; }
};

➡
int main()
{
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```



```
template <typename T>
class Printer
{
public:
    void parentheses(const T& v) const { std::cout << '(' << v << ')'; }
    void quote(const T& v) const { std::cout << '"' << v << '"'; }
};
```



```
int main()
{
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```

le compilateur va générer le type
Printer<double> à partir du
modèle = instantiation

```
template <typename T>
class Printer
{
public:
    void parenthe
    void quote(co
};
```

```
int main()
{
```

```
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```

```
class Printer<double>
{
public:
    Printer() {} // constructeur généré par défaut
};
```

les fonctions-membres sont
générées uniquement au moment
de leur utilisation

```
template <typename T>
class Printer
{
public:
    void parenthe
    void quote(co
};
```

```
int main()
{
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```

```
class Printer<double>
{
public:
    Printer() {}
    void quote(const double& v) const { ... }
};
```

les fonctions-membres sont
générées uniquement au moment
de leur utilisation

Une spécialisation permet, pour une liste d'arguments de template spécifiques, d'utiliser un autre modèle de code que celui du template au moment de l'instanciation.

```
template <int Diviseur>
void divide_by(int dividende)
{
    std::cout << (dividende / Diviseur) << std::endl;
}
```

```
template <>
void divide_by<0>(int dividende)
{
    std::cerr << "Cannot divide " << dividende << " by 0!" << std::endl;
}
```

Une spécialisation permet, pour une liste d'arguments de template spécifiques, d'utiliser un autre modèle de code que celui du template au moment de l'instanciation.

template

```
template <int Diviseur>
void divide_by(int dividende)
{
    std::cout << (dividende / Diviseur) << std::endl;
}
```

```
template <>
void divide_by<0>(int dividende)
{
    std::cerr << "Cannot divide " << dividende << " by 0!" << std::endl;
}
```

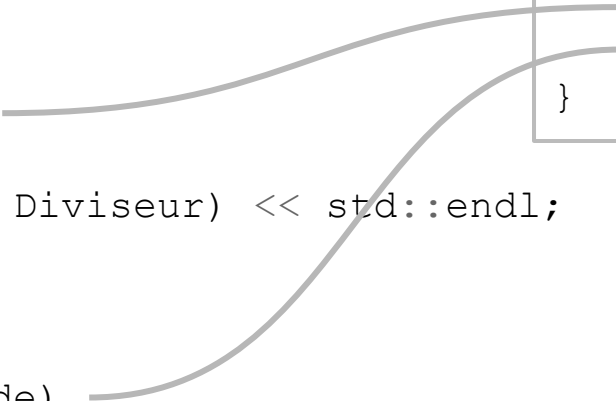
spécialisation pour <0>

Une spécialisation permet, pour une liste d'arguments de template spécifiques, d'utiliser un autre modèle de code que celui du template au lieu de l'instanciation.

```
template <int Diviseur>
void divide_by(int dividende)
{
    std::cout << (dividende / Diviseur) << std::endl;
}
```

```
template <>
void divide_by<0>(int dividende)
{
    std::cerr << "Cannot divide " << dividende << " by 0!" << std::endl;
}
```

```
int main()
{
    divide_by<3>(5);
    divide_by<0>(5);
}
```



Lorsqu'on spécialise une fonction-template, il faut indiquer les valeurs pour **TOUS** les paramètres de template. On parle de **spécialisation totale**.

```
template <typename R, typename T1, typename T2>
```

```
R add(T1 v1, T2 v2)
```

```
{
```

```
    return static_cast<R>(v1 + v2);
```

```
}
```

```
template <>
```

```
std::string add<std::string, const char*, const char*>(const char* str1,  
                                                         const char* str2)
```

```
{
```

```
    return std::string { str1 } + str2;
```

```
}
```

Lorsqu'on spécialise une fonction-template, il faut indiquer les valeurs pour **TOUS** les paramètres de template. On parle de **spécialisation totale**.

```
template <typename R, typename T1, typename T2>
```

```
R add(T1 v1, T2 v2)
```

```
{
```

```
    return static_cast<R>(v1 + v2);
```

```
}
```

on n'a plus aucun paramètre de template

```
template <>
```

```
std::string add<std::string, const char*, const char*>(const char* str1,  
                                                         const char* str2)
```

```
{
```

```
    return std::string { str1 } + str2;
```

```
}
```


Lorsqu'on spécialise une fonction-template, il faut indiquer les valeurs pour **TOUS** les paramètres de template. On parle de **spécialisation totale**.

```
template <typename R, typename T1, typename T2>
```

```
R add(T1 v1, T2 v2)
```

```
{
```

```
    return static_cast<R>(v1 + v2);
```

```
}
```

```
template <>
```

```
std::string add<std::string, const char*, const char*>(const char* str1,  
                                                         const char* str2)
```

```
{
```

```
    return std::string { str1
```

```
}
```

on a bien spécifié les valeurs
des 3 paramètres ici

On peut spécialiser une classe-template de trois manières différentes :

- spécialisation totale de classe-template
- spécialisation partielle de classe-template
- spécialisation de fonction-membre (forcément totale, puisque c'est une spécialisation de fonction)

Lorsqu'on spécialise une classe-template, il faut réécrire **l'intégralité** de la classe-template, pas uniquement les morceaux qui nous intéressent (attributs + fonctions-membres + implémentation de ces fonctions).

Cela permet d'adapter le type d'attributs à un cas donné (par exemple, pour optimiser le code, ou gérer des cas particuliers). `std::vector<bool>` est une spécialisation de `std::vector`, car l'implémentation peut-être optimisée en passant par des masques de bits.

Exemple : <https://godbolt.org/z/GG4ed91Kc>

On peut également faire des spécialisations partielles. Contrairement aux spécialisations totales dans lesquelles on s'attend à ce que les premiers chevrons de la spécialisation soit vides, on peut laisser une partie des paramètres non spécifiées, voir en utiliser certains pour construire les paramètres de template finaux (par exemple : “je veux que ma spécialisation concerne tous les `std::vector<qqch>`”, `qqch` est un paramètre de template de la spécialisation)

Exemple 1 : <https://godbolt.org/z/vczrsfeqP>

Exemple 2 : <https://godbolt.org/z/TM1W7shGz>


On peut également spécialiser seulement certaines fonctions d'une classe-template. Dans ce cas, la spécialisation doit être totale, puisqu'il s'agit d'une spécialisation de fonctions-template.

Une valeur, c'est le résultat d'une expression. Une expression peut-être une variable, un littéral ou encore un appel de fonction.

```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```

Une valeur, c'est le résultat d'une expression. Une expression peut être une variable, un littéral, un appel de fonction ou encore le résultat d'une opération.

```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```



3 et 5 sont des
expressions

Une valeur, c'est le résultat d'une expression. Une expression peut-être une variable, un littéral ou encore un appel de fonction.

```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```

l'appel à `min<int>` est
aussi une expression

Une valeur, c'est le résultat d'une expression. Une expression peut-être une variable, un littéral ou encore un appel de fonction.

```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```

cout et endl sont
également des expressions

Une valeur, c'est le résultat d'une expression. Une expression peut-être une variable, un littéral ou encore un appel de fonction.

```
int main()
```

```
{
```

```
    auto a = std::min(3, 5);
```

```
    std::cout << a << std::endl;
```

```
}
```

l'appel à l'opérateur <<
constitue une expression

Une valeur, c'est le résultat d'une expression. Une expression peut-être une variable, un littéral ou encore un appel de fonction.

```
int main()
```

```
{
```

```
    auto a = std::min(3, 5);
```

```
    std::cout << a << std::endl;
```

```
}
```

l'appel à l'opérateur <<
constitue une expression

Une valeur, c'est le résultat d'une expression. Une expression peut-être une variable, un littéral ou encore un appel de fonction.

```
int main()
{
    auto a = std::min(3, 5);
    std::cout << a << std::endl;
}
```

ici, a n'est pas une
expression (définition
de variable)

Une valeur, c'est le résultat d'une expression. Une expression peut-être une variable, un littéral ou encore un appel de fonction.

```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```

cette partie là n'est pas non plus
une expression (précédence de
l'opérateur <<)

Lorsqu'on évalue une expression, on doit considérer 3 points :

- la valeur de l'expression
- la catégorie de cette valeur
- les éventuels effets de bord de l'expression

Ce qui va nous intéresser ici, c'est la catégorie de valeur.

On a deux catégories principales :

- les l-values : les valeurs sont stockées dans un emplacement mémoire bien défini
- les r-values : il s'agit de littéraux, résultats temporaires, et autres valeurs dont l'emplacement mémoire n'est pas bien défini

```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```

l-value ou r-value ?


```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```

r-value !

```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```

I-value ou r-value ?

```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```



l-value !

```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```

I-value ou r-value ?

```
int main()  
{  
    auto a = std::min(3, 5);  
    std::cout << a << std::endl;  
}
```



l-value !

```
int get_min(int& v1, int& v2)
{
    return std::min(v1, v2);
}

int main()
{
    int v1 = 3;
    int v2 = 6;
    std::cout << get_min(v1, v2) << std::endl;
}
```

```
int get_min(int& v1, int& v2)
{
    return std::min(v1, v2);
}

int main()
{
    int v1 = 3;
    int v2 = 6;
    std::cout << get_min(v1, v2) << std::endl;
}
```

l-value !

l-value !

r-value !

on retourne maintenant
par référence

```
int& get_min(int& v1, int& v2)
{
    return std::min(v1, v2);
}

int main()
{
    int v1 = 3;
    int v2 = 6;
    std::cout << get_min(v1, v2) << std::endl;
}
```


on retourne maintenant
par référence

```
int& get_min(int& v1, int& v2)
{
    return std::min(v1, v2);
}
```

```
int main()
{
    int v1 = 3;
    int v2 = 6;
    std::cout << get_min(v1, v2) << std::endl;
}
```

l-value !

- l-value \Rightarrow binder à une adresse en mémoire. On peut écrire `&exp` pour récupérer l'adresse en mémoire de l'expression.
(`exp = ...`)
- r-value \Rightarrow littéraux, résultats temporaires, etc. Si on écrit `&exp`, ça ne compile pas.
`exp = ...` compilera jamais

L'intérêt de savoir catégoriser les valeurs, c'est qu'on peut faire de l'overloading avec.

On pourra donc choisir d'appeler une fonction plutôt qu'une autre si on lui passe une l-value ou une r-value.

Pour indiquer qu'une fonction attend une r-value, il faut utiliser une double esperluette : `int&&` par exemple.

```
void print_value_category(int& v)
{
    std::cout << v << " l-value" << std::endl;
}

void print_value_category(int&& v)
{
    std::cout << v << " r-value" << std::endl;
}

int main()
{
    int v = 3;
    print_value_category(5);
    print_value_category(v);
}
```

```
void print_value_category(int& v)
{
    std::cout << v << " l-value" << std::endl;
}
```

```
void print_value_category(int&& v)
{
    std::cout << v << " r-value" << std::endl;
}
```

```
int main()
{
    int v = 3;
    print_value_category(5);
    print_value_category(v);
}
```



r-value

```
void print_value_category(int& v)
{
    std::cout << v << " l-value" << std::endl;
}
```

```
void print_value_category(int&& v)
{
    std::cout << v << " r-value" << std::endl;
}
```

```
int main()
{
    int v = 3;
    print_value_category(5);
    print_value_category(v);
}
```

I-value

```
void print_value_category(int& v)
{
    std::cout << v << " l-value" << std::endl;
}
```

r-value → ne compile pas,
parce que une r-value n'est pas
convertible en l-value non-const

```
int main()
{
    int v = 3;
    print_value_category(5);
    print_value_category(v);
}
```

```
void print_value_category(const int& v)
{
    std::cout << v << " l-value" << std::endl;
}
```

```
int main()
{
    int v = 3;
    print_value_category(5);
    print_value_category(v);
}
```

r-value → compile parce que
une r-value est convertible en
l-value const


```
void print_value_category(int&& v)
{
    std::cout << v << " r-value" << std::endl;
}
```

```
int main()
{
    int v = 3;
    print_value_category(5);
    print_value_category(v);
}
```

I-value → erreur de compilation
si pas d'overload I-value

```
void print_value_category(int&& v)
{
    std::cout << v << " r-value" << std::endl;
}
```

```
int main()
{
    int v = 3;
    print_value_category(5);
    print_value_category(std::move(v));
}
```

std::move permet de convertir
une l-value en r-value

Lorsqu'une fonction attend un objet par l-value, la fonction indique qu'elle souhaite utiliser l'objet. Elle pourra ou non muter son état, mais l'ownership de ses ressources est préservé.

En revanche, lorsqu'une fonction attend un objet par r-value, elle indique qu'elle compte en fait lui retirer l'ownership de ses ressources.

Soit une classe Object. On définit :

- **le constructeur de déplacement** comme le constructeur qui attend un `Object&&` en paramètre
- **l'opérateur d'assignation par déplacement** comme l'operator= qui attend un `Object&&` en paramètre

```
class Object
{
public:
    Object(Object&& other)
    : _values { std::move(other._values) }
    {}

    Object& operator=(Object&& other)
    {
        if (this != &other)
        {
            _values = std::move(other._values);
        }

        return *this;
    }

private:
    std::vector<int> _values;
};
```

```
class Object
```

```
{  
public:  
    Object(Object&& other)  
        : _values { std::move(other._values) }  
    {}  
  
    Object& operator=(Object&& other)  
    {  
        if (this != &other)  
        {  
            _values = std::move(other._values);  
        }  
  
        return *this;  
    }  
}
```

on vole le contenu de other

on vole le contenu de other

```
private:  
    std::vector<int> _values;  
};
```

Selon vous, que fait le constructeur de déplacement de `std::unique_ptr` ? et son opérateur d'assignation par déplacement ?

Testons : <https://godbolt.org/z/bKr4x8M7v>