



# Chapitre 8

## Généricité



1. Introduction et exemples [3-9]
2. Fonctions génériques (patrons de fonctions) [10-46]
3. Classes génériques (patrons de classes) [47-64]
4. Alias de types génériques (C++11) et variables génériques (C++14) [65-68]
5. Contrats et concepts dans la STL [69-76]
6. Résumé [77-78]



# 1. Introduction et exemples



# Introduction : exemples de généricité

Le concept de **généricité** ne vous est pas inconnu :

(« générique » vient de général, qui est le contraire de « spécifique » ou particulier)

- La **surcharge de fonctions** permet de définir la même fonction pour des paramètres différents
- Les classes **string**, **wstring**, **u16string**, **u32string** définissent similairement des chaînes de caractères différentes (**char**, **wchar\_t**, **char16\_t**, **char32\_t**)
- Les classes **vector** et **array** peuvent contenir divers types de données
- Les fonctions de la bibliothèque **algorithm** s'appliquent à des conteneurs variés



# Exemple 1 (chapitre 5)

- Vecteurs contenant des éléments de divers types
  - un seul type par vecteur
  - vector est une classe
  - possède divers constructeurs

*Comment la définir pour pouvoir écrire `vector<T>` ?*

```
vector<int>    v1;  
vector<int>    v2(3);  
vector<int>    v3(5, 7);  
vector<int>    v4{1, 2};  
vector<double> v5(9);  
vector<string> v6(4, "Hi");  
vector<string> v7(6);
```

```
vector<int> v{2, 3, 5, 7, 11};  
v.push_back(13); // {2,3,5,7,11,13}  
v.resize(3);     // {2,3,5}  
v.pop_back();    // {2,3}  
v.resize(6, 1);  // {2,3,1,1,1,1}  
v.clear();       // {}  
v.push_back(42); // {42}  
v.resize(5);     // {42,0,0,0,0}
```



## Exemple 2 (chapitre 2)

- Les fonctions fournies par la librairie `<limits>` comme `numeric_limits<TYPE>::lowest()` ou `numeric_limits<TYPE>::max()` où on remplace TYPE par `short`, `int`, `unsigned`, etc.

Type T	<code>numeric_limits&lt;T&gt;::lowest()</code>	<code>numeric_limits&lt;T&gt;::max()</code>
<code>signed char</code>	-128	127
<code>signed short int</code>	-32768	32767
<code>signed int</code>	-2147483648	2147483647
<code>signed long int</code>	comme <code>int</code> ou <code>long long</code>	
<code>signed long long int</code>	-9223372036854775808	9223372036854775807

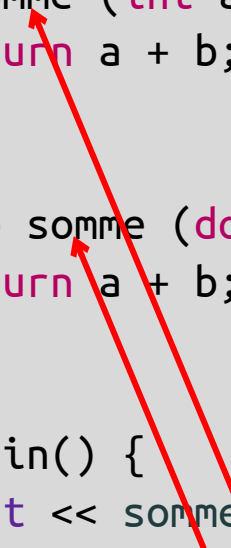
*Comment faire pour pouvoir écrire `numeric_limits<int>` ?*

- Attention, les signes '`<`' et '`>`' n'ont pas la même signification pour les classes ou fonctions génériques que pour les librairies ajoutées avec `#include <...>`

## Exemple 3 : surcharge de fonction

- Plusieurs fonctions peuvent partager **le même nom** à condition que leurs **profils** – le nombre et l'ordre des types des paramètres – permettent au compilateur de déterminer quelle version appeler
- Si le code de ces fonctions est quasiment identique (sauf les types), beaucoup de **code se trouve dupliqué** 📌

```
int somme (int a, int b) {  
    return a + b;  
}  
  
double somme (double a, double b) {  
    return a + b;  
}  
  
int main() {  
    cout << somme(10, 20) << endl;  
    cout << somme(1.0, 1.5) << endl;  
}
```



# Amélioration de l'exemple 3



- Pour éviter cette duplication, la fonction somme peut être écrite de manière **générique**, valable pour tout type T

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

- Le code client devient

```
int main() {
    cout << somme<int>(10, 20) << endl;
    cout << somme<double>(1.0, 1.5) << endl;
}
```

- ❖ On simplifiera plus tard l'écriture des appels grâce à la **déduction de types**





# Généricité : syntaxe générale C++

- Syntaxe générale d'une déclaration générique

```
template < liste_de_paramètres > déclaration
```

- Elle permet de déclarer génériquement une famille de :
  - fonctions (y compris des fonctions membres)
  - classes
  - variables (à partir de C++14)
  - ou un alias à une famille de types (à partir de C++11)
- La généricité est un élément clé de la C++ *Standard Library*, puisque celle-ci est issue de la *Standard Template Library (STL)*



## 2. Fonctions génériques (patrons de fonctions)



- La déclaration et la définition d'une fonction générique sont :
  - précédées du mot réservé **template**
  - suivies des **noms de types génériques**
    - placés **entre <>**
    - chacun précédé du mot réservé **typename**
- Les **noms de types** de la liste des paramètres génériques peuvent être utilisés comme tout autre type :
  - dans la déclaration (paramètres, type de retour)
  - dans le corps de la définition (variables, cast, ...)

```
// déclaration
template <typename T>
void echanger(T& v1, T& v2);

// définition
template <typename T>
void echanger(T& v1, T& v2) {
    T temp = v1;
    v1 = v2;
    v2 = temp;
}
```



# Écrire typename ou bien class ?

- Pour des raisons de rétrocompatibilité (avec des versions précédentes de la norme), on peut aussi écrire **class** à la place de **typename**
- En PRG1, on vous le déconseille
  - par exemple parce que 'int' n'est pas une classe

```
// déclaration
template <class T>
void echanger(T& v1, T& v2);

// définition
template <class T>
void echanger(T& v1, T& v2) {
    T temp = v1;
    v1 = v2;
    v2 = temp;
}
```



# Instanciation : implicite ou explicite

- La **définition d'une fonction générique** ne définit pas réellement une fonction, mais juste un **moule** devant être instancié
  - **compiler** un fichier qui ne contient que des définitions génériques **ne génère aucun code**
- Pour que le compilateur génère du code, il faut **instancier la fonction générique** avec des **types effectifs**
  1. *implicitement*, en appelant la fonction, spécifiant ou non les types
    - si les types ne sont pas spécifiés, ils seront *déduits* des arguments

```
int main() {  
    int a = 0, b = 1;  
    echanger<int>(a, b); //  
    // instanciation implicite,  
    // avec spécification du type,  
    // et appel de echanger<int>(int&, int&)  
}
```

1. ou *explicitement*, par une déclaration

```
template void echanger<int>(int&, int&);
```



- Il peut y avoir **plusieurs paramètres génériques**, séparés dans la liste par des virgules

```
template <typename T, typename U>  
void f(T v1, U v2) {  
    ...  
}
```

- Une telle fonction s'instancie en spécifiant les **types effectifs souhaités séparés par des virgules**, par exemple explicitement

```
template void f<int, double>(int, double);
```

- ❖ Si on indique moins de types dans l'instanciation ☾ déduction des arguments



Il n'est pas nécessaire de spécifier les types effectifs souhaités si ceux-ci peuvent être déduits du contexte.

La fonction générique suivante

```
template <typename T> void echanger(T& v1, T& v2);
```

peut aussi être instanciée explicitement ainsi (<> est optionnel)

```
template void echanger<>(int&, int&); // T=int est déduit  
template void echanger(char&, char&); // T=char est déduit
```

ou implicitement (dans ce cas, sans préciser le type) ainsi

```
int a = 0, b = 1;  
echanger<>(a, b); // deux versions possibles pour  
echanger(a, b);  // l'instanciation et l'appel de  
                  // echanger<int>(int&, int&)
```



# Déduction : nombre de types indiqués

- Dans l'initialisation implicite (par appel de la fonction), on peut spécifier ou non les valeurs des typename
  - toutes
  - seulement les premières
  - aucune (<> est alors optionnel)
- Les valeurs qui ne sont pas spécifiées seront déduites :

```
template <typename T, typename U> void f(T v1, U v2) { ... }
```

<code>f&lt;&gt;(a, b);</code>	→	T et U seront déduits des types de <code>a</code> et <code>b</code>
<code>f&lt;int&gt;(a, b);</code>	→	T = int, U sera déduit du type de <code>b</code> (et <code>a</code> peut être converti)
<code>f&lt;int, double&gt;(a, b);</code>	→	T = int, U = double (et <code>a</code> et <code>b</code> peuvent être convertis)





- Soit une fonction générique  $f$  de paramètres  $P_i$  et d'arguments génériques  $T_i$
- Soit un appel à  $f$  ne spécifiant pas explicitement tous les paramètres  $T_i$
- Pour chaque paire  $i$  d'arguments  $(A_i, P_i)$ , on déduit zéro, un ou plusieurs types  $T_j$  non spécifiés explicitement qui permettent que  $A_i$  et  $P_i$  soient le même type.
- La déduction est globalement possible si en combinant toutes ces paires :
  - tous les types  $T_i$  non spécifiés sont déduits
  - si plusieurs paires  $(A_i, P_i)$  déduisent un même argument générique  $T_j$ , c'est bien *le même type* qui est déduit par toutes les paires

```
template <typename T1, typename T2, typename T3>
void f(P1 p1, P2 p2, P3 p3, P4 p4);

int main() {
    A1 a1; A2 a2; A3 a3; A4 a4;
    f(a1, a2, a3, a4);
}
```

# HE<sup>VD</sup> IG Déduction : exemple



- Arguments génériques  $T_i$ :
  - $T_1 = T$
  - $T_2 = U$
- Paramètres de la fonction  $P_i$ 
  - $P_1 = \text{const vector}<T>\&$
  - $P_2 = \text{pair}<T, U>$
- Paramètres effectifs  $A_i$ 
  - $A_1(v) = \text{vector}<\text{int}>$
  - $A_2(p1) = \text{pair}<\text{int}, \text{double}>$
  - $A_2(p2) = \text{pair}<\text{double}, \text{int}>$

```
template <typename T, typename U>
void f(const vector<T>& v, pair<T, U> p);

int main() {
    vector<int> v;
    pair<int, double> p1;
    pair<double, int> p2;

    f(v, p1); // 1: vector<int> = vector<T>
              //      -> T = int, U non spécifié
              // 2: pair<int, double> == pair<T, U>
              //      -> T = int, U = double
              // 1 n 2: T = int, U = double -> OK

    f(v, p2); // 1: vector<int> == vector<T>
              //      T = int, U non spécifié
              // 2: pair<double, int> == pair<T, U>
              //      T = double, U = int
              // 1 n 2: impossible pour T
              //      -> erreur de compilation
}
```



# Déduction : pas de conversion de types

Attention, avec la déduction des paramètres génériques, c'est **toujours le type exact qui est passé**. Le compilateur ne peut pas déduire *et* convertir les types *en même temps* !

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

int main() {
    int i1, i2;
    double d1, d2;
    f(i1, i2);           // f<int>(int,int)
    f(d1, d2);           // f<double>(double,double)
    f(i1, d1);           // erreur de compilation
    f<int>(i1, d1);        // f<int>(int,int) avec conversion de d1 en int
    f<double>(i1, d1);     // f<double>(double,double) avec conversion de i1 en double
}
```



# Déduction : cas du type de retour

- La déduction n'est pas possible pour les arguments T qui ne sont pas l'un des paramètres de la fonction
- Tout argument non déductible doit être spécifié explicitement dans une instantiation
- Dans l'exemple ci-contre, seul le type From peut être déduit ; le type To doit être fixé explicitement en écrivant `<int>`
  - Note : si l'ordre des types génériques avait été inversé dans notre exemple (`<typename From, typename To>`), il aurait fallu donner à la fois From et To :

```
int i = convert<double, int>(d);
```

```
template <typename To, typename From>
To convert(const From& val) {
    return (To) val;
}

int main() {
    double d = 0.5;
    int i = convert<int>(d);
    // convert<int, double>(double);
    int i = convert<>(d);
    // ne compile pas
    ...
}
```



- On peut spécifier des valeurs par défaut pour les paramètres génériques

```
template <typename To = int, typename From = int>  
To convert(const From& val) {  
    return (To) val;  
}
```

- Mais cette possibilité est peu utilisée pour les fonctions génériques car la déduction d'arguments prime sur cette valeur par défaut

```
double d;  
convert(d);           // convert<int, double>  
convert<int>(d);       // convert<int, double>  
convert<int, int>(d);  // convert<int, int>
```



- Par contre, cela peut-être utile pour les paramètres non déductibles

```
template <typename From, typename To = float> // ordre inversé
To convert(const From& val) {
    return (To) val;
}
```

```
double d;
convert(d);           // convert<double, float>
convert<int>(d);       // convert<int, float>
convert<int, int>(d);  // convert<int, int>
```

Mais la surcharge de fonction permettra d'obtenir le même résultat plus naturellement.

- Ce sera en revanche largement utilisé pour les classes génériques. Par exemple `std::stack` est déclaré ainsi :

```
template <typename T, typename Container = std::deque<T>> class stack;
```



- Comme pour les fonctions, on peut surcharger les fonctions génériques de même nom en les distinguant par
  - Le nombre de paramètres (attention aux valeurs par défaut)
  - Le type de ces paramètres
  - Les références (&, const &, &&)
- On peut également donner le même nom à des fonctions simples et à des fonctions génériques

```
template <typename T>
void f(T);

template <typename T>
void f(T, T);

template <typename T>
void f(T, int);

template <typename T, typename U>
void f(T, U&);

template <typename T, typename U>
void f(T, const U&);

void f(int, float);
```



# Surcharge : résolution sans déduction

- Si l'on spécifie explicitement les paramètres génériques lors de l'appel de fonctions surchargées, alors la résolution suit exactement les mêmes règles que pour les fonctions simples (chap. 4)
- Par exemple, l'appel à `f<long>(i)`
  - doit choisir entre `f<long>(long&)` et `f<long>(const long&)`
  - sélectionne la deuxième parce qu'il n'y a pas de conversion de `int&` en `long&`

```
template <typename T> int f(T&) {  
    return 1;  
}  
  
template <typename T> int f(const T&) {  
    return 2;  
}  
  
int main() {  
    int i = 42;  
    cout << f<long>(i);    // 2  
    cout << f<int>(i);     // 1  
    cout << f<int>(42);    // 2  
}
```





# Surcharge : résolution avec déduction

- Si l'on utilise la déduction des paramètres génériques, il faut une règle supplémentaire
- Ci-contre, **f(v)** peut appeler
  - la fonction 1 avec T = `vector<int>`
  - la fonction 2 avec T = `int`
- La résolution de surcharge choisit la fonction 2 parce que son premier paramètre ( `vector<T>` ) est **plus spécialisé** que celui de la fonction 1 ( `T` )
- Le paramètre générique P1 est **plus spécialisé** que P2
  - si pour tout paramètre effectif a qui permet à f(a) d'appeler f<T>(P1),
  - f(a) peut aussi appeler f<T>(P2),
  - mais pas le contraire

```
template <typename T>
int f(T) { return 1; }

template <typename T>
int f(vector<T>) { return 2; }

int main() {
    vector<int> v(42);

    cout << f<vector<int>>(v); // 1
    // seule callable si T = vector<int>

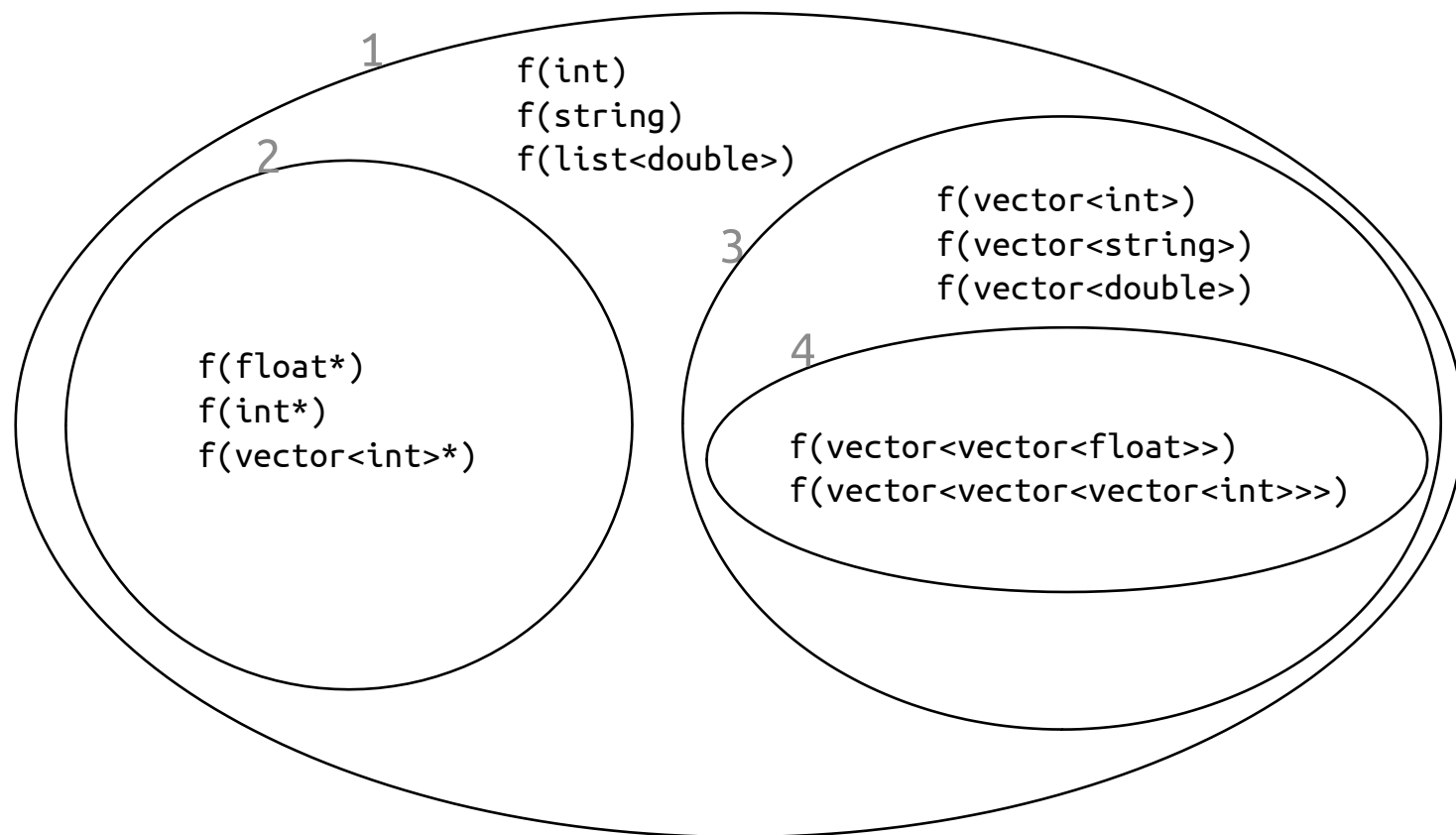
    cout << f<int>(v);          // 2
    // seule callable si T = int

    cout << f(v);              // 2
    // les 2 fonctions sont appelables,
    // mais P2 est plus spécialisé que P1
}
```



# Ordre partiel « plus spécialisé »

```
template <typename T> void f(T) {...}           // 1
template <typename T> void f(T*) {...}          // 2
template <typename T> void f(vector<T>) {...}    // 3
template <typename T> void f(vector<vector<T>>) {...} // 4
```

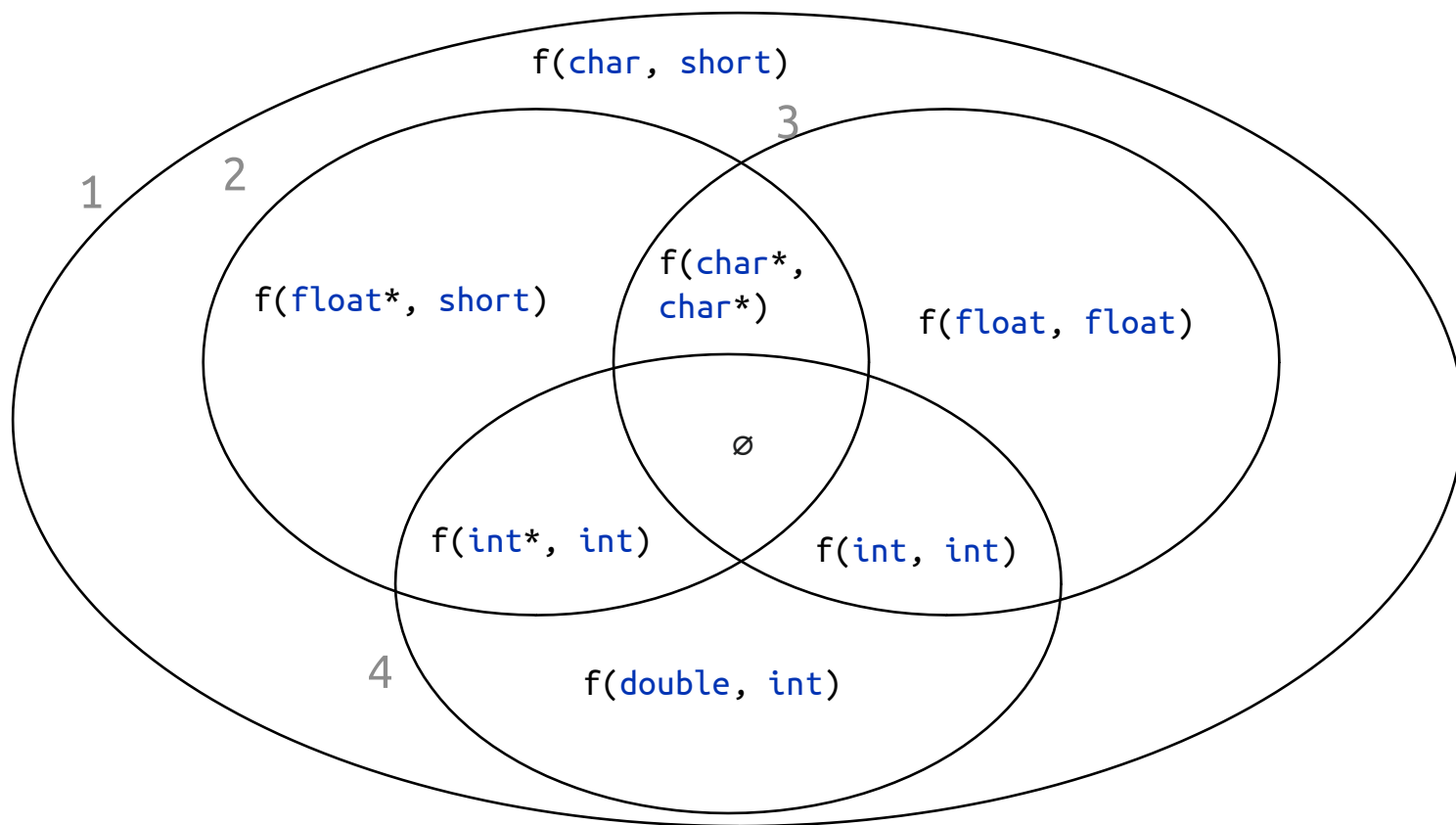


- « f1 est plus spécialisé que f2 » signifie que l'ensemble des paramètres effectifs capables d'appeler f1 « est un sous-ensemble de » celui de f2
- 2, 3, et 4 sont plus spécialisées que 1 parce que toute fonction qui peut les appeler peut aussi appeler 1
- 4 est plus spécialisée que 3 parce que toute fonction qui peut l'appeler peut aussi appeler 3
- Il n'y a pas d'ordre « plus spécialisé »
  - Ni entre 2 et 3, ni entre 2 et 4
  - Mais cela ne pose pas de problème de résolution, leurs intersections étant vides



# Ordre partiel avec plusieurs paramètres

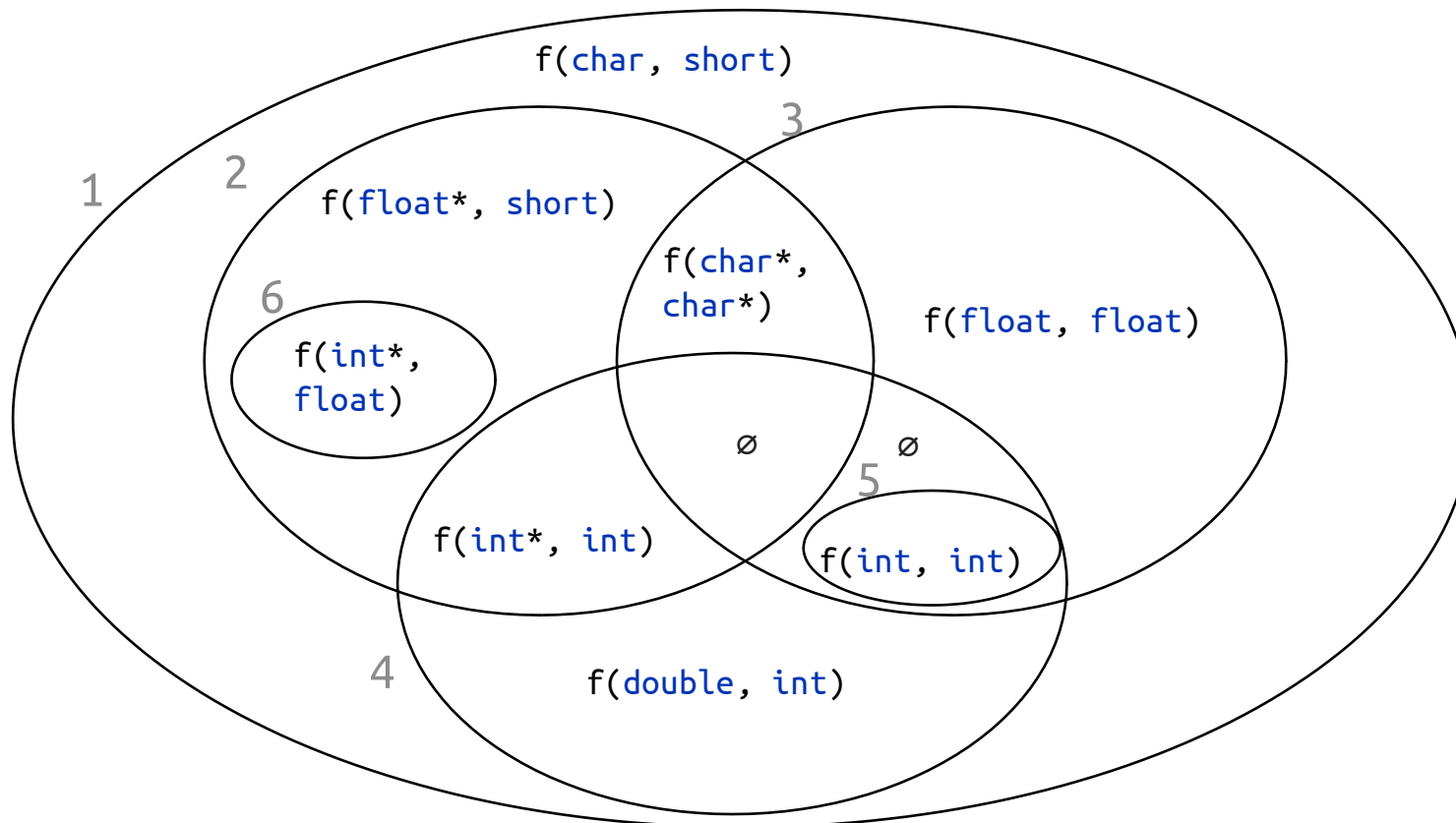
```
template <typename T, typename U> void f(T, U) {...} // 1
template <typename T, typename U> void f(T*, U) {...} // 2
template <typename T> void f(T, T) {...} // 3
template <typename T> void f(T, int) {...} // 4
```



- Quand  $f$  a plusieurs paramètres, des ensembles non-inclus l'un dans l'autre peuvent avoir une intersection non vide
- Ces intersections seront des lieux d'ambiguïté
- Par exemple,  $f(\text{char}^*, \text{char}^*)$ 
  - peut appeler 1, 2 ou 3
  - 2 et 3 sont plus spécialisées que 1, mais il n'y a pas d'ordre entre elles
  - l'appel est ambigu
- Par contre,  $f(\text{float}^*, \text{short})$ 
  - peut appeler 1 ou 2
  - 2 est plus spécialisé que 1
  - 2 est donc appelée



```
template <typename T, typename U> void f(T, U) {...} // 1
template <typename T, typename U> void f(T*, U) {...} // 2
template <typename T> void f(T, T) {...} // 3
template <typename T> void f(T, int) {...} // 4
void f(int, int) {...} // 5
void f(int*, float) {...} // 6
```



- On peut ajouter les fonctions non-générique au diagramme de Venn
- Une fonction non-générique est toujours « plus spécialisée » qu'une fonction générique
- Une fonction non-générique peut résoudre une ambiguïté
  - `f(int, int)` appelle 5
  - Sans la fonction 5, il y aurait ambiguïté entre 3 et 4



1. Établir la liste des fonctions viables (génériques ou pas) en tenant compte des ...
  1. **Nom** de la fonction (y.c. visibilité du namespace) \* Si l'appel de la fonction est du type  $f<>(\dots)$  ou  $f<\text{type}(s)>(\dots)$ , seules les fonctions génériques sont considérées.
  2. **Nombre de paramètres** (exact ou plus grand avec paramètres par défaut)
  3. Type **exact ou conversion** possible pour les paramètres *non-génériques*
  4. **Déduction d'arguments** pour les paramètres *génériques* non spécifiés explicitement
2. S'il y a plusieurs candidates, déterminer si une est « meilleure que toutes les autres »
  1. Au sens de l'algorithme de résolution de surcharge du chapitre 4
    1. individuellement pour chaque paramètre : type exact > promotion > ajustement
    2. puis intersection des choix des paramètres
  2. Si 2.1 ne détermine pas d'ordre entre 2 fonctions, au sens de l'ordre partiel « plus spécialisé », sachant qu'une fonction non-générique est plus spécialisée qu'une fonction générique

Si l'algorithme sélectionne une unique fonction, elle est appelée. S'il sélectionne 0 ou plusieurs fonctions, il y a erreur de compilation.



# Exemple 1 : `int i, j; f(i, j);`

```
template <typename T, typename U> void f(T, U) {...} // 1
template <typename T, typename U> void f(T*, U) {...} // 2
template <typename T> void f(T, T) {...} // 3
template <typename T> void f(T, int) {...} // 4
void f(int, int) {...} // 5
void f(int*, float) {...} // 6
```

## Étape 1 : fonctions viables par déduction des arguments génériques ou conversion des paramètres non génériques

R1 := 1<sup>er</sup> paramètre = `int`

R2 := 2<sup>ème</sup> paramètre = `int`

R1  $\cap$  R2

1 : `<T,U> = <int,?>`

2 : pas de déduction possible

3 : `<T> = <int>`

4 : `<T> = <int>`

5 : type exact

6 : pas de conversion

`<T,U> = <?,int>`

`<T,U> = <?,int>`

`<T> = <int>`

`<T> = <?>`, type exact

type exact

ajustement `int -> float`

`<T,U> = <int,int>`

pas callable

`<T> = <int>`

`<T> = <int>`

callable

pas callable



# Exemple 1 : `int i, j; f(i, j);`

```
template <typename T, typename U> void f(T, U) {...} // 1 void f<int,int>(int, int);
template <typename T, typename U> void f(T*, U) {...} // 2 pas callable
template <typename T> void f(T, T) {...} // 3 void f<int>(int, int);
template <typename T> void f(T, int) {...} // 4 void f<int>(int, int);
void f(int, int) {...} // 5
void f(int*, float) {...} // 6
```

## Étape 2.1 : résolution au sens du chapitre 4

R1 := 1<sup>er</sup> paramètre = `int`

R2 := 2<sup>e</sup> paramètre = `int`

1 : **type exact**  
2 : N/A  
3 : **type exact**  
4 : **type exact**  
5 : **type exact**  
6 : N/A

**type exact**  
N/A  
**type exact**  
**type exact**  
**type exact**  
N/A

R1 = { 1, 3, 4, 5 }

R2 = { 1, 3, 4, 5 }

$R1 \cap R2 = \{ 1, 3, 4, 5 \}$

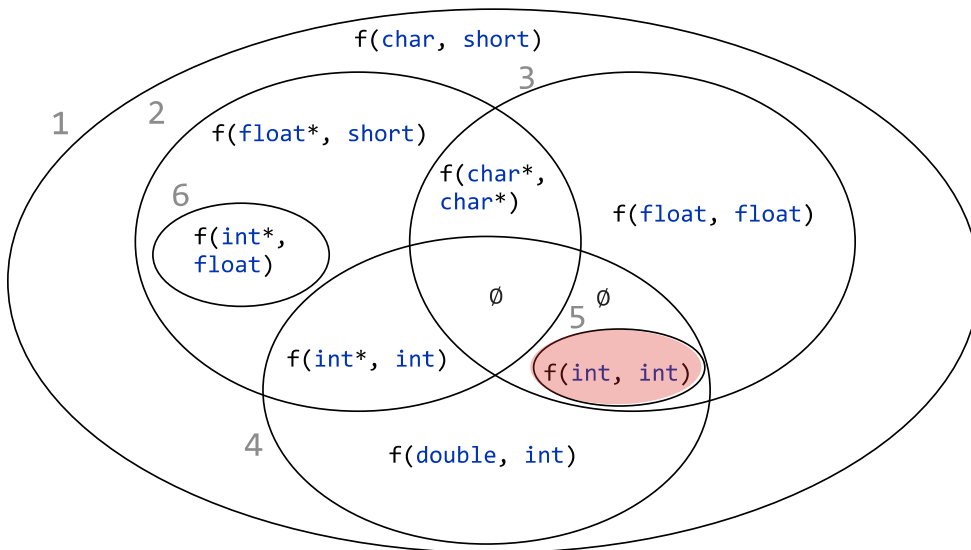


# Exemple 1 : `int i, j; f(i, j);`

```
template <typename T, typename U> void f(T, U) {...}
template <typename T, typename U> void f(T*, U) {...}
template <typename T> void f(T, T) {...}
template <typename T> void f(T, int) {...}
void f(int, int) {...}
void f(int*, float) {...}
```

// Déductions :  
// 1 void f<int,int>(int, int);  
// 2 pas callable  
// 3 void f<int>(int, int);  
// 4 void f<int>(int, int);  
// 5  
// 6

## Étape 2.2 : ordre partiel dans $R1 \cap R2 = \{ 1, 3, 4, 5 \}$



- 5 est non-générique, ce qui la rend plus spécialisée que les fonctions génériques 1, 3 et 4
- `f(i,j);` appelle donc 5





## Exemple 2 : `char c; int i; f(c, i);`

```
template <typename T, typename U> void f(T, U) {...} // 1
template <typename T, typename U> void f(T*, U) {...} // 2
template <typename T> void f(T, T) {...} // 3
template <typename T> void f(T, int) {...} // 4
void f(int, int) {...} // 5
void f(int*, float) {...} // 6
```

### Étape 1 : fonctions viables par déduction des arguments génériques ou conversion des paramètres non génériques

R1 := 1<sup>er</sup> paramètre = `char`

R2 := 2<sup>e</sup> paramètre = `int`

R1  $\cap$  R2

1 : `<T,U> = <char,?>`

2 : pas de déduction possible

3 : `<T> = <char>`

4 : `<T> = <char>`

5 : promotion `char -> int`

6 : pas de conversion

`<T,U> = <?,int>`

`<T,U> = <?,int>`

`<T> = <int>`

`<T> = <?>`, type exact

type exact

ajustement `float -> int`

`<T,U> = <char,int>`

pas callable

pas callable

`<T> = <char>`

callable

pas callable



## Exemple 2 : `char c; int i; f(c, i);`

```
template <typename T, typename U> void f(T, U) {...} // 1 void f<char,int>(char, int);
template <typename T, typename U> void f(T*, U) {...} // 2 pas callable
template <typename T> void f(T, T) {...} // 3 pas callable
template <typename T> void f(T, int) {...} // 4 void f<char>(char, int);
void f(int, int) {...} // 5
void f(int*, float) {...} // 6
```

### Étape 2.1 : résolution au sens du chapitre 4

R1 := 1<sup>er</sup> paramètre = `char`

R2 := 2<sup>e</sup> paramètre = `int`

1 : **type exact**

2 : N/A

3 : N/A

4 : **type exact**

5 : promotion `char` -> `int`

6 : N/A

**type exact**

N/A

N/A

**type exact**

**type exact**

N/A

R1 = { 1, 4 }

R2 = { 1, 4, 5 }

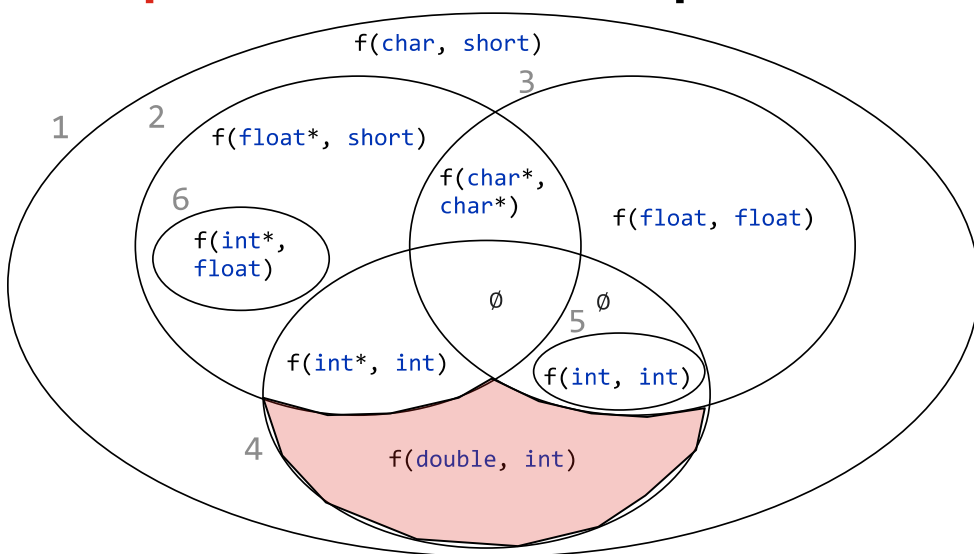
R1 ∩ R2 = { 1, 4 }



## Exemple 2 : `char c; int i; f(c, i);`

```
template <typename T, typename U> void f(T, U) {...} // 1 void f<char,int>(char, int);
template <typename T, typename U> void f(T*, U) {...} // 2 pas callable
template <typename T> void f(T, T) {...} // 3 pas callable
template <typename T> void f(T, int) {...} // 4 void f<char>(char, int);
void f(int, int) {...} // 5
void f(int*, float) {...} // 6
```

### Étape 2.2 : ordre partiel dans $R1 \cap R2 = \{ 1, 4 \}$



- 4 est plus spécialisée que 1
- `f(c,i);` appelle donc 4



## Exemple 3 : `int *p,*q; f(p, q);`

```
template <typename T, typename U> void f(T, U) {...} // 1
template <typename T, typename U> void f(T*, U) {...} // 2
template <typename T> void f(T, T) {...} // 3
template <typename T> void f(T, int) {...} // 4
void f(int, int) {...} // 5
void f(int*, float) {...} // 6
```

### Étape 1 : fonctions viables par déduction des arguments génériques ou conversion des paramètres non génériques

R1 := 1<sup>er</sup> paramètre = `int*`

R2 := 2<sup>e</sup> paramètre = `int*`

R1  $\cap$  R2

1 : `<T,U> = <int*,?>`

2 : `<T,U> = <int,?>`

3 : `<T> = <int*>`

4 : `<T> = <int*>`

5 : pas de conversion

6 : type exact

`<T,U> = <?,int*>`

`<T,U> = <?,int*>`

`<T> = <int*>`

pas de conversion

pas de conversion

pas de conversion

`<T,U> = <int*,int*>`

`<T,U> = <int,int*>`

`<T> = <int*>`

pas callable

pas callable

pas callable



## Exemple 3 : `int *p,*q; f(p, q);`

```
template <typename T, typename U> void f(T, U) {...} // 1 void f<int*,int*>(int*, int*);
template <typename T, typename U> void f(T*, U) {...} // 2 void f<int,int*>(int*,int*);
template <typename T> void f(T, T) {...} // 3 void f<int*>(int*,int*);
template <typename T> void f(T, int) {...} // 4 pas appellable
void f(int, int) {...} // 5
void f(int*, float) {...} // 6
```

### Étape 2.1 : résolution au sens du chapitre 4

R1 := 1<sup>er</sup> paramètre = `int*`

R2 := 2<sup>ème</sup> paramètre = `int*`

1 : **type exact**

2 : **type exact**

3 : **type exact**

4 : N/A

5 : N/A

6 : N/A

**type exact**

**type exact**

**type exact**

N/A

N/A

N/A

R1 = { 1, 2, 3 }

R2 = { 1, 2, 3 }

R1  $\cap$  R2 = { 1, 2, 3 }

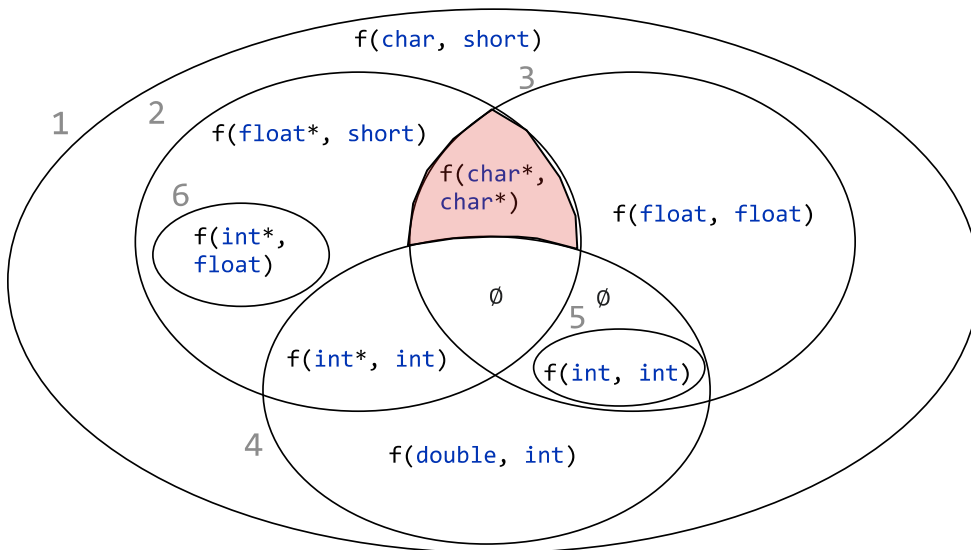


## Exemple 3 : `int *p,*q; f(p, q);`

```
template <typename T, typename U> void f(T, U) {...}
template <typename T, typename U> void f(T*, U) {...}
template <typename T> void f(T, T) {...}
template <typename T> void f(T, int) {...}
void f(int, int) {...}
void f(int*, float) {...}
```

// Déductions :  
// 1 void f<int\*,int\*>(int\*, int\*);  
// 2 void f<int,int\*>(int\*,int\*);  
// 3 void f<int\*>(int\*,int\*);  
// 4 pas appellable  
// 5  
// 6

### Étape 2.2 : ordre partiel dans $R1 \cap R2 = \{ 1, 2, 3 \}$



- 2 est plus spécialisé que 1
- 3 est plus spécialisé que 1
- Il n'y a pas d'ordre entre 2 et 3
- L'appel est ambigu



On peut redéfinir spécifiquement une fonction générique pour un argument générique donné

- en utilisant `template<>`
  - note : `<>` est obligatoire
- suivi de la fonction où **tous les types\*** sont spécifiés

\* pas de spécialisation partielle pour les fonctions génériques

```
template <typename T>
bool estDeTypeInt(T) {
    return false;
}

template<>
bool estDeTypeInt(int) {
    return true;
}

int main() {
    cout << boolalpha
        << estDeTypeInt(1) << ' ' // true
        << estDeTypeInt('a') << ' ' // false
        << estDeTypeInt(1.) << endl ; // false
}
```



# Rappel : instantiation explicite ou implicite

- **Implicite** : en *appelant* simplement la fonction (ici, en précisant le type du paramètre typename comme `<int>`)

```
int main() {  
    int a = 0, b = 1;  
    echanger<int>(a, b); // instantiation ET appel de  
                        // echanger<int>(int&, int&)  
}
```

- **Implicite avec déduction de type** : `echanger(a, b)`  
souvent possible, mais voir les restrictions plus loin
- **Explicite** : *déclarations* successives typées

```
template void echanger<int>(int&, int&);  
template void echanger<char>(char&, char&);
```



# HE<sup>VD</sup> IG Compilation séparée



La *définition d'une fonction générique* ne définit pas réellement une fonction, mais un moule ☾ deux possibilités en terme de compilation séparée

1. Placer la *définition dans un fichier header*, sans utiliser de déclaration séparée, ni de fichier .cpp
  - tout code qui inclut ce header peut instancier la fonction (en général implicitement, par son appel) avec tous types d'arguments
2. Placer
  - la *déclaration* dans un fichier header
  - la *définition* dans un fichier .cpp accompagnée des instantiations explicites de tous les types qui seront utiles (alors aucun autre type ne sera utilisable)



- L'instanciation *implicite* des templates rend la tâche du compilateur complexe dans un cadre de compilation séparée
- Chaque instanciation implicite dans un fichier .cpp entraîne la génération de code dans le fichier objet correspondant
  - ce code doit être nettoyé par l'éditeur de liens
- Pour simplifier et accélérer la compilation, on peut indiquer au compilateur qu'une instance est déjà définie ailleurs
  - comme une instanciation explicite, mais avec le mot-clé **extern**

```
extern template void echanger<int>(int&, int&);  
extern template void echanger<char>(char&, char&);
```
  - à utiliser typiquement au début de  $n - 1$  des fichiers .cpp qui sont inclus dans le main.cpp et qui utilisent la fonction générique
  - si la définition n'est pas trouvée : erreur d'édition de liens



# Paramètres génériques autres que types

- Comme nous l'avons déjà vu avec la classe `std::array`, il est possible d'avoir des paramètres génériques qui ne sont pas des types. Par exemple

```
template <int N> void incr(int& i) {  
    i += N;  
}  
  
int main() {  
    int i = 1;  
    incr<10>(i); // i vaut 11  
    ...  
}
```

- Attention, « `int N` » ressemble à une variable, mais il s'agit d'une valeur constante qui est doit être déterminée à la compilation, et non à l'exécution
- ❖ Note : un tel paramètre générique est parfois appelé paramètre expression



# Nature des paramètres autres que types

Ces paramètres peuvent être des constantes des types suivants seulement :

- Un type **intégral** (`bool`, `char` ... `int` ... `unsigned long long`)
  - Un type **énuméré**
  - Un **pointeur vers ou une référence à**
    - une fonction
    - un objet alloué statiquement
    - un membre statique (objet ou fonction) d'une classe
- ❖ Il faut que leur valeur puisse être déterminée à la compilation



- Une version assez particulière de « Hello, World! »

```
template <std::string& temp> void g() {  
    temp += "World!";  
}  
  
std::string s; // variable globale dont on peut déterminer une  
               // référence à la compilation  
  
int main() {  
    s = "Hello, ";  
    g<s>();  
    cout << s << endl;  
}
```



- Une méthode originale pour calculer une factorielle avec ...
  - paramètre générique non-type N
  - spécialisation pour N = 0

```
template <unsigned N> constexpr unsigned factorielle() {  
    return N * factorielle<N-1>();  
}  
  
template<> constexpr unsigned factorielle<0>() {  
    return 1;  
}  
  
int main() {  
    constexpr unsigned F7 = factorielle<7>(); // 5040  
}
```



### **3. Classes génériques (patrons de classes)**



- Déclaration d'une classe générique

```
template < liste_de_paramètres > déclaration
```

- Par exemple, on peut rendre générique la classe CVector (PRG1, chap. 7) pour les types des coordonnées

```
class CVector {  
    double x,y;  
public:  
    CVector() {};  
    CVector(double a, double b)  
        : x(a), y(b) {}  
};
```

```
template <typename T>  
class CVector {  
    T x, y;  
public:  
    CVector() {}  
    CVector(T a, T b)  
        : x(a), y(b) {}  
};
```





- Une classe générique doit être **instanciée** pour que le compilateur génère son code (comme les fonctions)
  - on peut l'instancier **explicitement**
- ou **implicitement**, en l'utilisant dans le code, p.ex. dans le `main()`

```
template class CVector<char>;
```

```
CVector<int> v(1,2);  
CVector<double> w(1,2);
```



- Avant C++17,
  - toujours indiquer les paramètres génériques entre <>
  - pas de déduction d'arguments pour les classes, contrairement aux fonctions
- Depuis C++17,
  - déduction possible des arguments à partir des *paramètres passés au constructeur* selon les même règles que pour les fonctions génériques
  - Le concepteur d'une classe peut guider cette déduction (hors sujet pour PRG1)
- Pour éviter l'instanciation implicite : mot-clé **extern**

```
extern template class CVector<int>;
```



- Dans une classe générique, les fonctions membres peuvent être définies de deux façons, et utilisent les variables dans `typename` et le nom de la classe ainsi :

## 1. définition en ligne

```
template <typename T> class CVector {  
    ...  
    T produitScalaire(const CVector<T>& cv) const {  
        return x * cv.x + y * cv.y;  
    }  
    ...  
};
```

- ## 2. séparation de la *déclaration* et *définition* pour les fonctions membres non triviales (slide suivante) = la bonne pratique



# Déclaration et définition séparées

- On **déclare** la méthode dans la déclaration de la classe

```
template <typename T>
class CVector {
    ...
    T produitScalaire(const CVector<T>& cv) const;
    ...
};
```

- On la **définit** en dehors de cette déclaration, avec **obligation** d'indiquer la classe générique (patron) à laquelle elle appartient, comme suit :

```
template <typename T>
T CVector<T>::produitScalaire(const CVector<T>& cv) const {
    return x * cv.x + y * cv.y;
}
```



- Cela vaut également pour les opérateurs membres qui sont (sur ce point) des fonctions membres comme les autres

```
template <typename T> class CVector {  
    ...  
    CVector<T> operator+(const CVector<T>& cv) const;  
    ...  
};
```

```
template <typename T>  
CVector<T> CVector<T>::operator+(const CVector<T>& cv) const {  
    CVector<T> temp;  
    temp.x = x + cv.x;  
    temp.y = y + cv.y;  
    return temp;  
}
```



## La définition d'une classe générique

1. Ne peut pas être compilée telle quelle pour donner du code objet
  - doit être **instanciée** (comme pour les fonctions génériques)
2. Doit plutôt être considérée comme une déclaration
  - il faut l'inclure entièrement dans un fichier header
  - y compris les fonctions membres

### CVector.h

```
#ifndef CVECTOR_H
#define CVECTOR_H

template <typename T>
class CVector {
    T x, y;
public:
    CVector() {}
    CVector(T a, T b): x(a), y(b) {}
    T produitScalaire(const CVector<T>& cv) const;
    CVector<T> operator+(const CVector<T>& cv) const;
};

template <typename T>
T CVector<T>::produitScalaire(const CVector<T>& cv) const {
    return x * cv.x + y * cv.y;
}

template <typename T>
CVector<T> CVector<T>::operator+(const CVector<T>& cv) const {
    CVector<T> temp;
    temp.x = x + cv.x;
    temp.y = y + cv.y;
    return temp;
}

#endif
```



# Compilation séparée : solution courante

- couper le fichier header en deux : déclarations + définitions
- inclure le fichier avec les définitions après les déclarations

## CVector.h

```
#ifndef CVECTOR_H
#define CVECTOR_H

template <typename T>
class CVector {
    T x, y;
public:
    CVector<T>() {}
    CVector<T>(T a, T b) : x(a), y(b) {}
    T produitScalaire(const CVector<T>& cv) const;
    CVector<T> operator+(const CVector<T>& cv) const;
};

#include "CVectorImpl.h"

#endif
```

## CVectorImpl.h

```
#ifndef CVECTORIMPL_H
#define CVECTORIMPL_H

template <typename T>
T CVector<T>::produitScalaire(const CVector<T>& cv) const {
    return x * cv.x + y * cv.y;
}

template <typename T>
CVector<T> CVector<T>::operator+(const CVector<T>& cv) const {
    CVector<T> temp;
    temp.x = x + cv.x;
    temp.y = y + cv.y;
    return temp;
}

#endif
```



- Comme les autres fonctions, les fonctions membres d'une classe peuvent aussi être génériques pour d'autres types
- Attention à choisir des noms différents pour les **paramètres génériques** de la classe et de la fonction membre (ici, T et U)
- La définition s'écrit alors avec deux mots-clés `template`

```
template <typename T>
class CVector {
    ...
    template <typename U>
    CVector<U> convert();
    ...
};
```

```
template <typename T>
template <typename U>
CVector<U> CVector<T>::convert() {
    return CVector<U>((U)x, (U)y);
}
```





- Pour pouvoir afficher les objets de type `CVector<T>`, il faut surcharger `operator<<` de manière générique aussi :

```
// déclaration avancée de CVector, pour pouvoir le
// mentionner dans la déclaration de operator<< juste après
template <typename T> class CVector;

// déclaration + définition de l'opérateur <<
template <typename T>
ostream& operator<<(ostream& os, const CVector<T>& cv) {
    return os << cv.x << ' ' << cv.y;
}

template <typename T> class CVector {
    // amitié entre CVector<T> et l'opérateur << générique avec
    // le paramètre générique effectif T
    friend ostream& operator<< <T>(ostream& os, const CVector<T>& cv);
    ...
};
```



- Notons que les déclarations d'amitié peuvent s'écrire de plusieurs manières

```
friend ostream& operator<< <T>(ostream& os, const CVector<T>& cv);  
friend ostream& operator<< <T>(ostream& os, const CVector& cv);  
friend ostream& operator<< <>(ostream& os, const CVector<T>& cv);  
friend ostream& operator<< <>(ostream& os, const CVector& cv);
```

- Mais si l'on oublie <> ou <T> avant la parenthèse ouvrante, alors on aura une erreur de compilation



- Pour traiter différemment un type spécifique, on peut **spécialiser** :
  - certaines méthodes, ou
  - toute la classe générique
- On le fait en les **redéfinissant** pour ce type précis
- Par exemple, on peut spécialiser le produit scalaire pour le type `bool` ainsi :

```
template <typename T> // définition générale
T CVector<T>::produitScalaire(const CVector<T>& cv) const {
    return x * cv.x + y * cv.y;
}
```

```
template<> // définition spécialisée
bool CVector<bool>::produitScalaire(const CVector& cv) const {
    return (x and cv.x) or (y and cv.y);
}
```



- Pour spécialiser toute une classe générique, on doit réécrire l'entièreté de la classe pour un argument (type) particulier

```
template<> class CVector<bool> {  
    // réécriture de toute la classe pour le type bool  
};
```



- Pour les classes à plusieurs paramètres génériques, il est possible de ne les spécialiser que partiellement, en gardant certains paramètres génériques

```
template<typename T1, typename T2, int I>
class A {};                                // template primaire

template<typename T, int I>
class A<T, T*, I> {};                      // #1: spécialisation partielle.
                                           // T2 est un pointeur vers T1

template<typename T, typename T2, int I>
class A<T*, T2, I> {};                     // #2: spécialisation partielle.
                                           // T1 est un pointeur

template<typename T>
class A<int, T*, 5> {};                     // #3: spécialisation partielle.
                                           // T1 est un int, I vaut 5,
                                           // et T2 est un pointeur

template<typename X, typename T, int I>
class A<X, T*, I> {};                      // #4: spécialisation partielle.
                                           // T2 est un pointeur
```



- Considérons le problème suivant.  
On dispose des conteneurs génériques Liste et Tableau :

```
template <typename T> class Liste { ... };  
template <typename T> class Tableau { ... };
```

- On peut utiliser un Tableau pour créer une 3<sup>e</sup> sorte de conteneur : Pile

```
template <typename T> class Pile {  
    Tableau<T> data;  
    ...  
};
```

- Mais pourrait-on laisser à l'utilisateur le choix d'utiliser Liste ou Tableau pour créer une Pile ? Il faudrait passer le type du conteneur comme paramètre générique.

# Paramètres template template



- Une solution (choisie par la STL) consiste à ajouter un paramètre générique pour indiquer le type du conteneur :

```
template <typename T, typename Conteneur> class Pile {  
    Conteneur data;  
    ...  
};
```

- Pour utiliser cette classe, on doit déclarer les piles ainsi :

```
Pile<int, Tableau<int>> p1;  
Pile<double, Liste<double>> p2;
```

- Mais on préférerait passer directement le type au conteneur (pour déduire `Tableau<int>` du premier `int`) et écrire ceci :

```
Pile<int, Tableau> p1;  
Pile<double, Liste> p2;
```

# HE<sup>VD</sup> IG C'est possible !



- Modifier la déclaration de `Pile` en utilisant un nouveau type de paramètre générique : `template template (!)`

```
template <typename T, template <typename> class Conteneur>
class Pile {
    Conteneur<T> data;
    ...
};
```

- Au prix d'une déclaration plus complexe de `Pile`
  - son utilisation est simplifiée
  - le risque de se tromper en mélangeant les types disparaît (par exemple en écrivant `Pile<double, Tableau<int>> p1;`)
- Attention, c'est le seul cas où il faut utiliser le mot-clé `class` et pas `typename` (du moins avant la norme C++17)





## **4. Alias de types génériques (C++11) et variables génériques (C++14)**



- C++11 introduit les alias de types génériques. Ils s'écrivent

```
template < template-parameter-list >  
    using identifier = type-id ;
```

- Par exemple :

```
template <typename T> using Tab10 = array<T,10>;  
template <typename T> using ptr = T*;
```

- Utilisation :

```
ptr<int> pi;           // au lieu de : int* pi;  
Tab10<double> tab;    // au lieu de : array<double,10> tab;
```



- Pour simplifier des types complexes, par exemple :

```
template <typename T>  
using rIter = vector<T>::reverse_iterator;
```

Error: Missing 'typename' prior to dependent type  
name 'vector<T>::reverse\_iterator'

Message  
explicite !

- Le compilateur ne peut être sûr que reverse\_iterator est un type et non pas une variable, et vous suggère de le préciser avec le mot-clé **typename**

```
template <typename T>  
using rIter = typename vector<T>::reverse_iterator;
```

- Attention à ne mettre typename qu'en cas d'ambiguïté, sinon erreur de compilation.



- C++14 introduit la possibilité de déclarer des variables génériques. L'exemple classique est :

```
template <typename T>  
const T PI = T(3.1415926535897932385);
```

- On instancie et utilise par exemple cette variable ainsi :

```
template <typename T>  
T circular_area(T r) {  
    return PI<T> * r * r; // instantiation implicite de PI<T>  
}
```

- Cela permet d'éviter des conversions de types inutiles si nous avons défini PI de type `double` par exemple



## 5. Contrats et concepts dans la STL



- Écrire une fonction générique telle que

```
template <typename T> T square(T a) {  
    return a * a;  
}
```

Invalid operands to binary expression  
(`'const char *'` and `'const char *'`)

implique un contrat implicite avec l'utilisateur de cette fonction :

il doit l'instancier avec un type pour lequel l'opérateur `*` existe, par exemple le type `int`

```
cout << square(5) << endl; // affiche 25
```

- Par contre, le code suivant ne compile pas.

```
cout << square("Hello") << endl;
```

L'erreur de compilation est indiquée dans la fonction générique, pas là où elle est : i.e. à l'instanciation de `square` avec un type non compatible.



# Non-respect du contrat implicite

- Reprenons la version minimaliste de notre classe CVector

```
class CVector {  
    int x, y;  
public:  
    CVector(int x, int y) : x(x), y(y) {};  
};
```

- Et stockons des objets de ce type dans un std::vector.

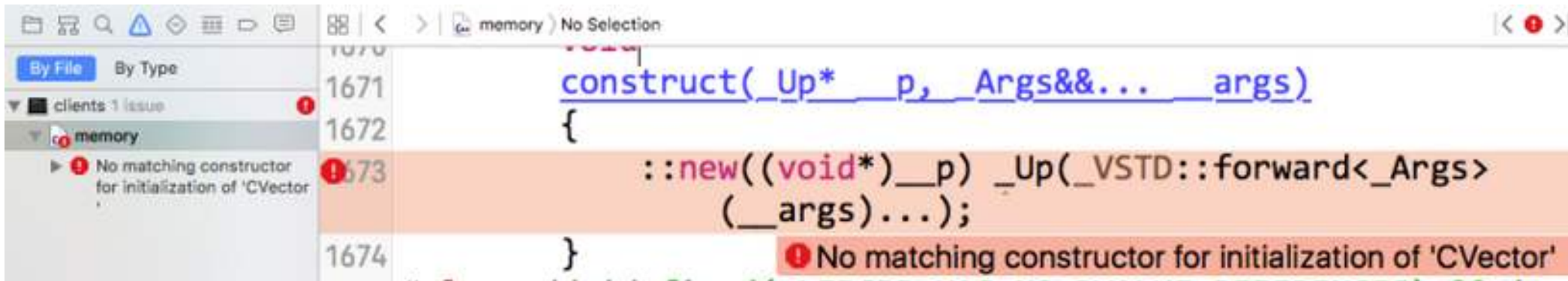
```
vector<CVector> v;
```

- Jusqu'ici tout va bien...
- Essayons de redimensionner ce vector 🐛 **Erreur !**

```
v.resize(2);
```

# Non-respect du contrat implicite

- Erreur de compilation à la ligne 1673 du fichier <memory> de la librairie standard !



```
1671 construct(_Up* __p, _Args&&... __args)  
1672 {  
1673     ::new((void*)__p) _Up(_VSTD::forward<_Args>  
    (__args)...);  
1674 }
```

❗ No matching constructor for initialization of 'CVector'

- Pourquoi ?





# Non-respect du contrat implicite

- Pour comprendre cette erreur, relisons le prototype de la méthode `resize`

public member function

**std::vector::resize**

C++98	C++11	?
<pre>void resize (size_type n, value_type val = value_type());</pre>		

- Il y a un 2<sup>e</sup> paramètre de type `value_type`. C'est en fait un typedef du paramètre générique `T` de `vector`

## Member types

C++98	C++11	?
member type		definition
<code>value_type</code>		The first template parameter ( <code>T</code> )

- `resize` avec le 2<sup>e</sup> paramètre non spécifié utilise donc le constructeur par défaut de `T`. Et ce constructeur n'existe pas pour notre classe `CVector`. Nous n'avons pas respecté le contrat implicite de `std::vector`



# Contrat explicite : `static_assert`

- Depuis C++11, on peut expliciter ce contrat avec l'expression

```
static_assert ( bool_constexpr , message )    (C++11)  
static_assert ( bool_constexpr )              (C++17)
```

- qui génère une erreur de compilation si l'expression booléenne est fausse.
  - L'expression doit être évaluable à la compilation
  - Elle utilisera souvent les fonctions de la librairie `<type_traits>` (hors sujet PRG1)

```
template <class T> class C {  
    public: static_assert(std::is_default_constructible<T>::value, "Bad T for C<T>");  
};  
  
class no_default {  
    public: no_default () = delete;  
};  
  
int main() {  
    C<no_default> c_error; // static assertion failed: Bad T for C<T>  
}
```



- Via des extensions depuis C++14 et dans les standards depuis C++20, on peut maintenant formaliser explicitement ces contrats via la notion de **concept**.

*(Wiki) : **Concepts** are an extension to the templates feature provided by the C++ programming language. Concepts are named Boolean predicates on template parameters, evaluated at compile time. A concept may be associated with a template (class template, function template, or member function of a class template), in which case it serves as a constraint: it limits the set of arguments that are accepted as template parameters.*

- Cette notion sort cependant du cadre de PRG1 et ne sera donc pas approfondie ici.
- Un exemple de mise en œuvre de cette notion de concept est toutefois proposée sur le slide suivant.



```
#include <iostream>
#include <string>
#include <concepts>
using namespace std;

template <typename T>
concept CanBeMultiplied = requires(T a, T b) {
    {a * b} -> T;
};

auto square(CanBeMultiplied a) {
    return a * a;
}

int main() {
    cout << square(5) << endl; // affiche 25
    cout << square("Hello") << endl; // error: cannot call function
                                         // 'auto square(auto:1) [with auto:1 = const char*]'
}
```



## 6. Résumé



- Fonctions génériques (patrons de fonctions)
  - permettent d'implémenter différemment une fonction selon le type de ses arguments : défis de la déduction des arguments, et de la résolution des surcharges
- Classes génériques (patrons de classes)
  - même idée pour les classes, mais sans déduction (< C++17)
- La généricité étend les capacités des classes, donc la POO
- Comprendre la généricité est important pour utiliser la STL
  - par exemple : <http://en.cppreference.com/w/cpp/container/vector/>
  - ou encore : <http://www.cplusplus.com/reference/vector/vector/>