

Fonctions Surchargées et généricité



Plan du chapitre



1. Surcharge de fonctions
2. Généricité (fonction)
3. Surcharge des opérateurs



1. Surcharge de fonctions



Surcharge de fonctions

- Surcharger une fonction, c'est utiliser le même nom pour des fonctions que l'on distingue uniquement par leurs paramètres
- La signature d'une fonction correspond aux caractéristiques de ses paramètres
 - Leur nombre et leur ordre
 - Leurs types, y compris la méthode de passage (valeur, référence ou référence constante)
- Le compilateur choisira la fonction à utiliser selon les paramètres effectifs (de l'appel) par rapport aux paramètres formels (du prototype) des fonctions candidates





Surcharge de fonctions

- Les éléments suivants **ne permettent pas** de différencier deux surcharges
 - Le type de **retour**
 - Les **valeurs par défaut** des paramètres
 - La présence d'un **const** pour un paramètre passé **par valeur**

```
int add(int lhs, int rhs=0) {  
    return lhs + rhs;  
}
```

```
short add(const int lhs, const int rhs) {  
    return short(lhs + rhs);  
}
```

- Si deux fonctions de même nom ne diffèrent que par l'un ou plusieurs éléments ci-dessus, le **compilateur signale une erreur à la déclaration** de la deuxième fonction

error: functions that differ only in
their return type cannot be overloaded



Quelles fonctions sont appelables ?

```
void f(int);           // 1
void f(double);        // 2
void f(const long &); // 3
void f(short &);       // 4
void f(const short &); // 5
void f(unsigned long); // 6
void f(unsigned long &); // 7
void f(unsigned);       // 8
void f(const unsigned &); // 9
void f(const string &); // 10
```

`f(int(0));`

- peut appeler les fonctions 1, 2, 3, 5, 6, 8, et 9
- mais pas 4, 7, et 10

`short s; f(s);`

- peut appeler les fonctions 1, 2, 3, 4, 5, 6, 8, et 9
- mais pas 7, et 10

`f("hello"s);`

- peut appeler la fonction 10
- mais aucune autre

`f(vector<int>(4));`

- ne peut appeler aucune fonction



Quelle surcharge est appelée ?

- 0 fonction appelable (`f(vector<int>(4))`) → erreur de compilation
- 1 fonction appelable (`f("hello"s)`) → elle est appelée
- 2+ fonctions appelables → laquelle choisir?
 - On considère successivement plusieurs critères de moins en moins stricts
 - Si le critère ne sélectionne aucune fonction, on passe à un critère moins strict
 - Si le critère sélectionne exactement 1 fonction → elle est appelée
 - Si le critère sélectionne plusieurs fonctions, l'appel est ambigu → erreur de compilation



Critère 1 : appel sans conversion ?

```
void f(int);           // 1
void f(double);        // 2
void f(const long &); // 3
void f(short &);      // 4
void f(const short &); // 5
void f(unsigned long); // 6
void f(unsigned long &); // 7
void f(unsigned);       // 8
void f(const unsigned &); // 9
void f(const string &); // 10
```

f(**int(0)**);

- ne peut appeler que 1 sans conversion

f(**double(0)**);

- ne peut appeler que 2 sans conversion

short s; f(s);

- ne peut appeler que 4 sans conversion
- Attention, pour appeler 5 il faudrait effectuer une *conversion simple* de s de **short** & à **const short &**

const short cs; f(cs);

- ne peut appeler que 5 sans conversion

f(**char(0)**);

- ne peut pas appeler sans conversion → critère suivant



```
void f(int);           // 1
void f(double);        // 2
void f(const long &); // 3
void f(short &);      // 4
void f(const short &); // 5
void f(unsigned long); // 6
void f(unsigned long &); // 7
void f(unsigned);       // 8
void f(const unsigned &); // 9
void f(const string &); // 10
```

`unsigned long ul; f(ul);`

- peut appeler 6 et 7 sans conversion → erreur de compilation

`const unsigned cu; f(cu);`

- peut appeler 8 et 9 sans conversion → erreur de compilation

■ Il ne faut jamais surcharger par valeur et par référence (6 et 7) ou par valeur et référence constante (8 et 9) du même type

■ Par contre, on peut surcharger par référence et référence constante (4 et 5)

Critère 2 : conversion simple



```
void f(int);           // 1
void f(double);        // 2
void f(const long &); // 3
void f(short &);      // 4
void f(const short &); // 5
void f(unsigned long); // 6
void f(unsigned long &); // 7
void f(unsigned);       // 8
void f(const unsigned &); // 9
void f(const string &); // 10
```

- Une conversion simple, c'est ...
 - Une variable transformée en constante (pas l'inverse)
 - Un tableau C transformé en pointeur (ou l'inverse)

long lo; f(lo);

- ne peut appeler que 3 par conversion simple



```
void f(int);           // 1
void f(double);        // 2
void f(const long &); // 3
void f(short &);      // 4
void f(const short &); // 5
void f(unsigned long); // 6
void f(unsigned long &); // 7
void f(unsigned);       // 8
void f(const unsigned &); // 9
void f(const string &); // 10
```

- Une promotion c'est ...
 - Un entier plus court que `int` converti en `int`
 - Un `float` converti en `double`
- `f(char(0));`
- ne peut appeler que 1 par promotion `char` → `int`
- `f(float(0));`
- ne peut appeler que 2 par promotion `float` → `double`
- `unsigned short us; f(us);`
- ne peut appeler que 1 par promotion `unsigned short` → `int`



```
void f(int);           // 1
void f(double);        // 2
void f(const long &); // 3
void f(short &);      // 4
void f(const short &); // 5
void f(unsigned long); // 6
void f(unsigned long &); // 7
void f(unsigned);       // 8
void f(const unsigned &); // 9
void f(const string &); // 10
```

- Une conversion de type c'est ...
 - toute conversion implicite qui existe du paramètre effectif vers le paramètre formel
 - C'est le critère minimal pour que la fonction soit appellable

f((long long)(0));
f((long double)(0));

- Peuvent appeler 1, 2, 3, 5, 6, 8, et 9 par conversion de type. Aucune fonction sans conversion, par conversion simple ou ajustement de type → appel ambigu

f("hello");

- Ne peut appeler que 10 par conversion de `const char*` → `string`



- Le compilateur recherche la **meilleure correspondance**, en testant **dans l'ordre**
 - **Correspondance exacte** - les types sont identiques
 - **Correspondance par conversion simple**
 - variable transformée en constante (pas l'inverse)
 - tableau transformé en pointeur (ou l'inverse)
 - **Correspondance par promotion numérique**
 - `bool`, `char` ou `short` => `int`
 - `float` => `double`
 - **Correspondance par conversion de type** (ajustement de type ou conversion dégradante) toutes celles acceptées par l'opérateur d'affectation
- Le compilateur **s'arrête au premier niveau** de correspondance trouvé
- Il y a **ambigüité** si plusieurs prototypes correspondent à ce niveau



- Notons qu'en C++, contrairement à Java, la notion de « meilleure » conversion implicite **n'existe pas**

```
void f(short n) {cout << "Appel de f(short n)" << endl;}
void f(long n) {cout << "Appel de f(long n)" << endl;}

int main() {
    int n = 1;
    f(n);      // Erreur à la compilation. Appel ambigu
}
```

- L'appel à `f(n)` est ambigu car la **conversion dégradante** `int -> short` est considérée au même niveau que l'**ajustement de type** `int -> long`



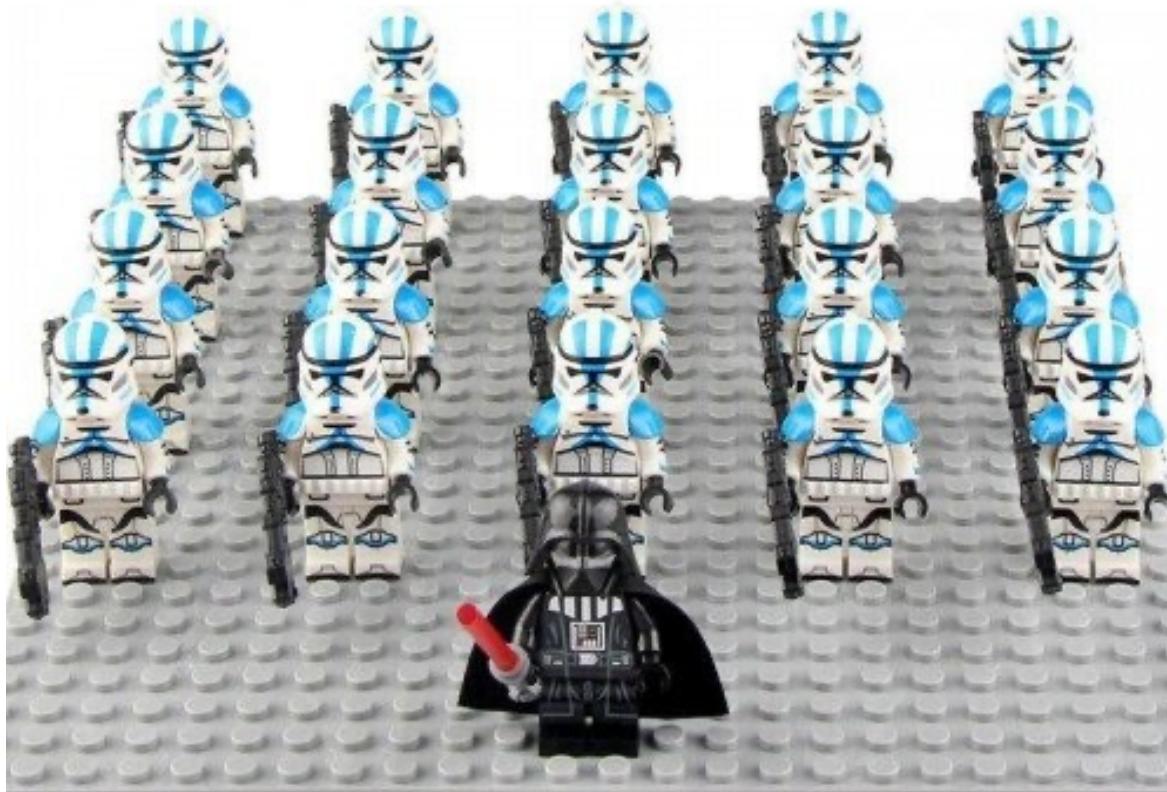
Fonctions à plusieurs paramètres

- Appliquer les 4 critères indépendamment pour chaque paramètre, en s'arrêtant au premier critère qui donne un ensemble non vide de fonctions
- Prendre l'intersection de ces ensembles. Si elle contient exactement 1 fonction, l'appel est non ambigu

```
void f(int, double); // 1
void f(int, int);    // 2
void f(char, double); // 3
```

- f(**short(0)**, **double(0)**);
 - p1 = {1,2}, p2 = {1,3}, p1 ∩ p2 = {1}
- f(**char(0)**, **char(0)**);
 - p1 = {1}, p2 = {2}, p1 ∩ p2 = {}, ambigu
- f(**long(0)**, **float(0)**);
 - p1 = {1,2,3}, p2 = {1,3}, p1 ∩ p2 = {1,3}, ambigu

2. Généricité (fonction)





Généricité – introduction

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



- 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}
```



Généricité – introduction

- 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<T>::lowest()</code>	<code>numeric_limits<T>::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 <...>`



- 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;  
}
```



- 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 – déclaration

- 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** (dès C++11)
- La généricité est un élément clé de la *C++ Standard Library*, puisque celle-ci est issue de la *Standard Template Library (STL)*



Généricité – déclaration

- 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;
}
```

Généricité – typename / 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;
}
```





Généricité – instantiation

- 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); //  
    // instantiation implicite,  
    // avec spécification du type,  
    // et appel de echanger<int>(int&, int&)  
}
```

2. *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



Généricité – paramètres

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&)
```



- Dans l'**instanciation 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) { ... }
```

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



Généricité – déduction

- 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);
}
```



Généricité – déduction

- 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 ∩ 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 ∩ 2: impossible pour T
               //      -> erreur de compilation
}
```



Généricité – déduction

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
```



- 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
}
```



Généricité – paramètre par défaut

- 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>
```



Généricité – paramètre par défaut

- 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;
```



Généricité – surcharge

- 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);
```



Généricité – résolution

- 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
}
```

Généricité – résolution



- Si l'on utilise la déduction des paramètres génériques, il faut une règle supplémentaire
- Ci-contre, **f(v)** peut appeler
 - la fonction 1 avec $T = \text{vector}<\text{int}>$
 - la fonction 2 avec $T = \text{int}$
- La résolution de surcharge choisit la fonction 2 parce que son premier paramètre ($\text{vector}<T>$) 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 appelable si T = vector<int>

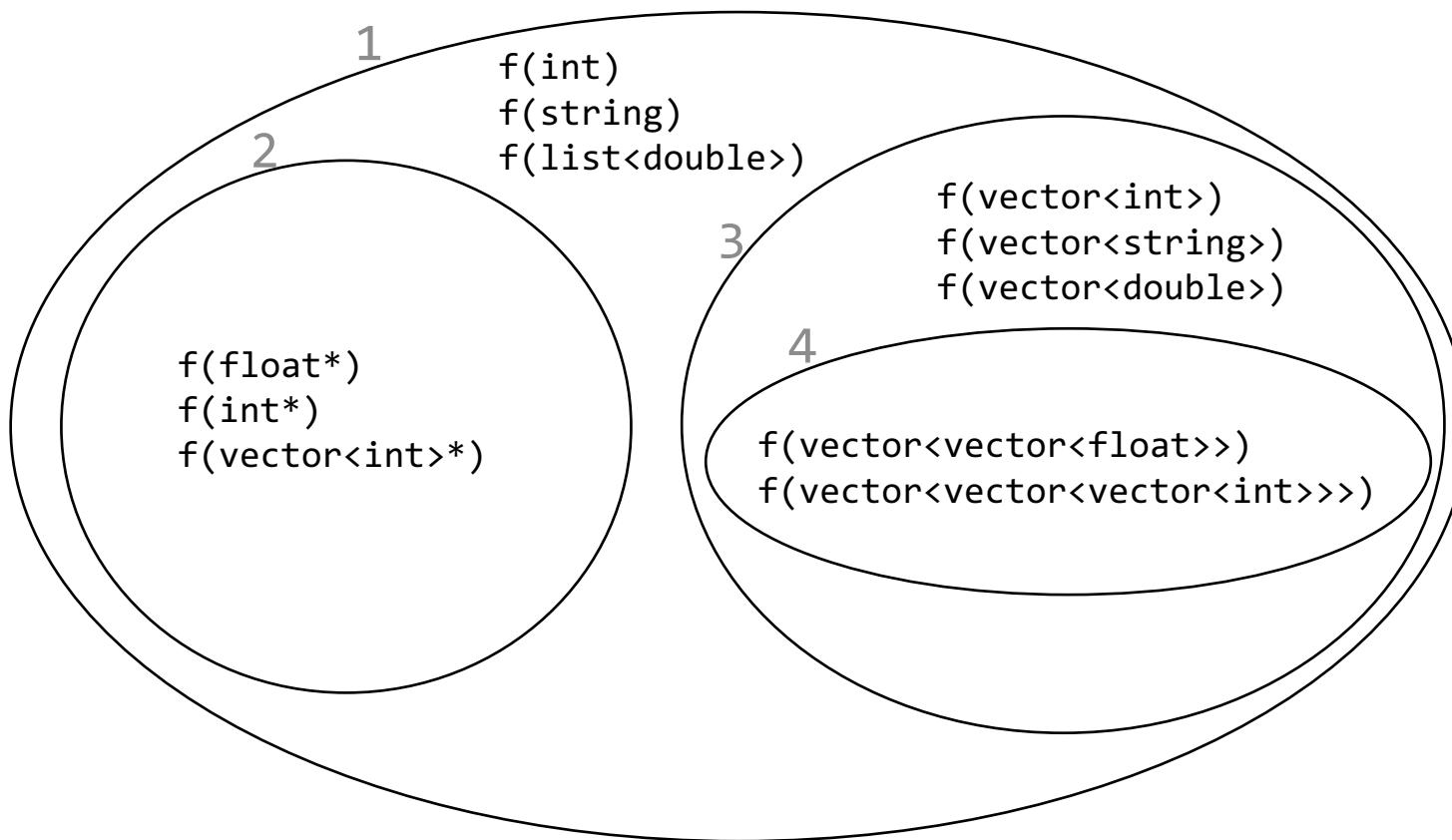
    cout << f<int>(v);           // 2
    // seule appelable si T = int

    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
```

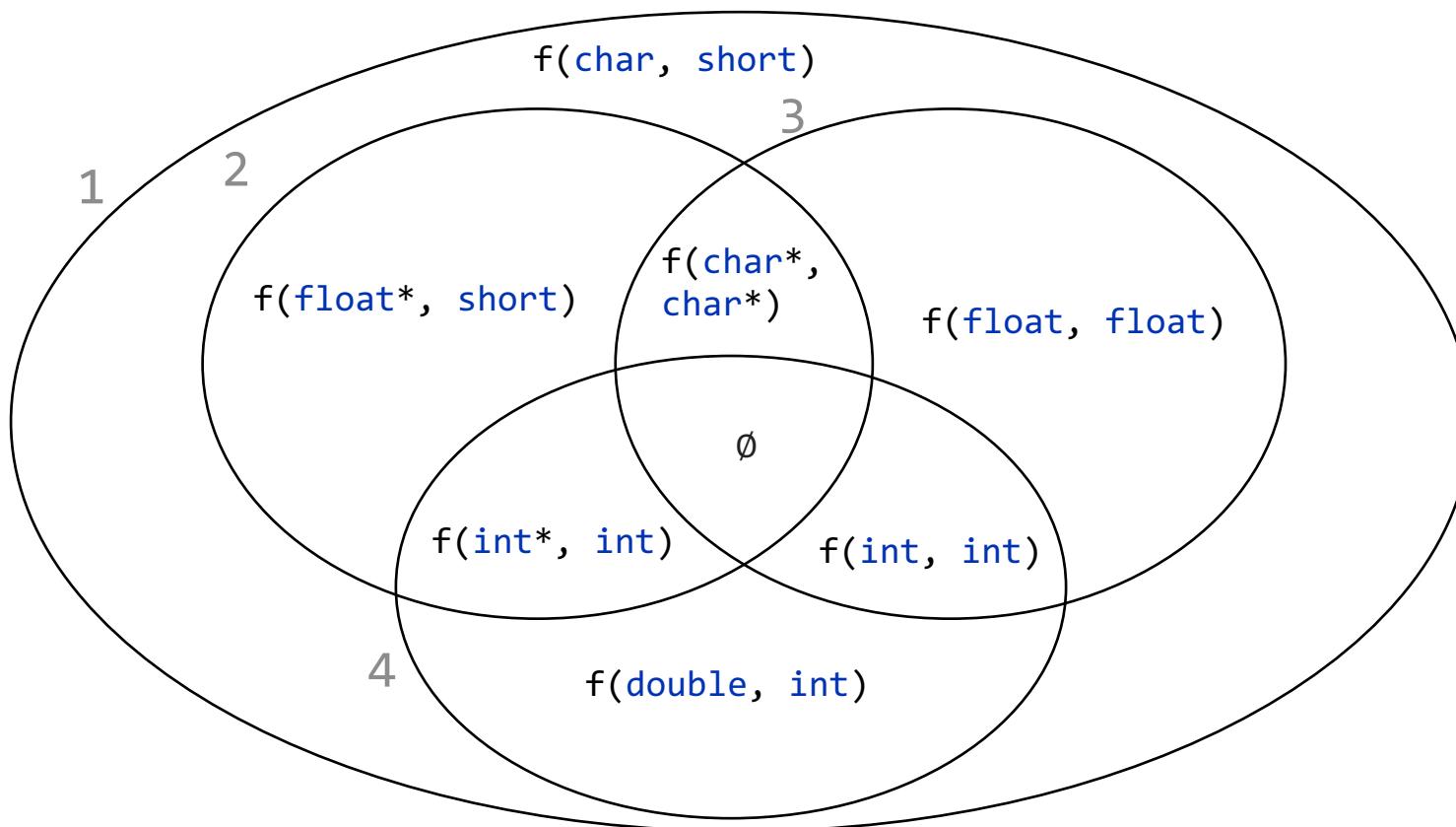


- « `f1` est plus spécialisé que `f2` » signifie que l'ensemble des paramètres effectifs capables d'appeler `f1` « est un sous-ensemble de » celui de `f2`
- 2, 3, et 4 sont plus spécialisées que 1 parce que toute fonction qui peut les appeler peut aussi appeler 1
- 4 est plus spécialisée que 3 parce 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
```

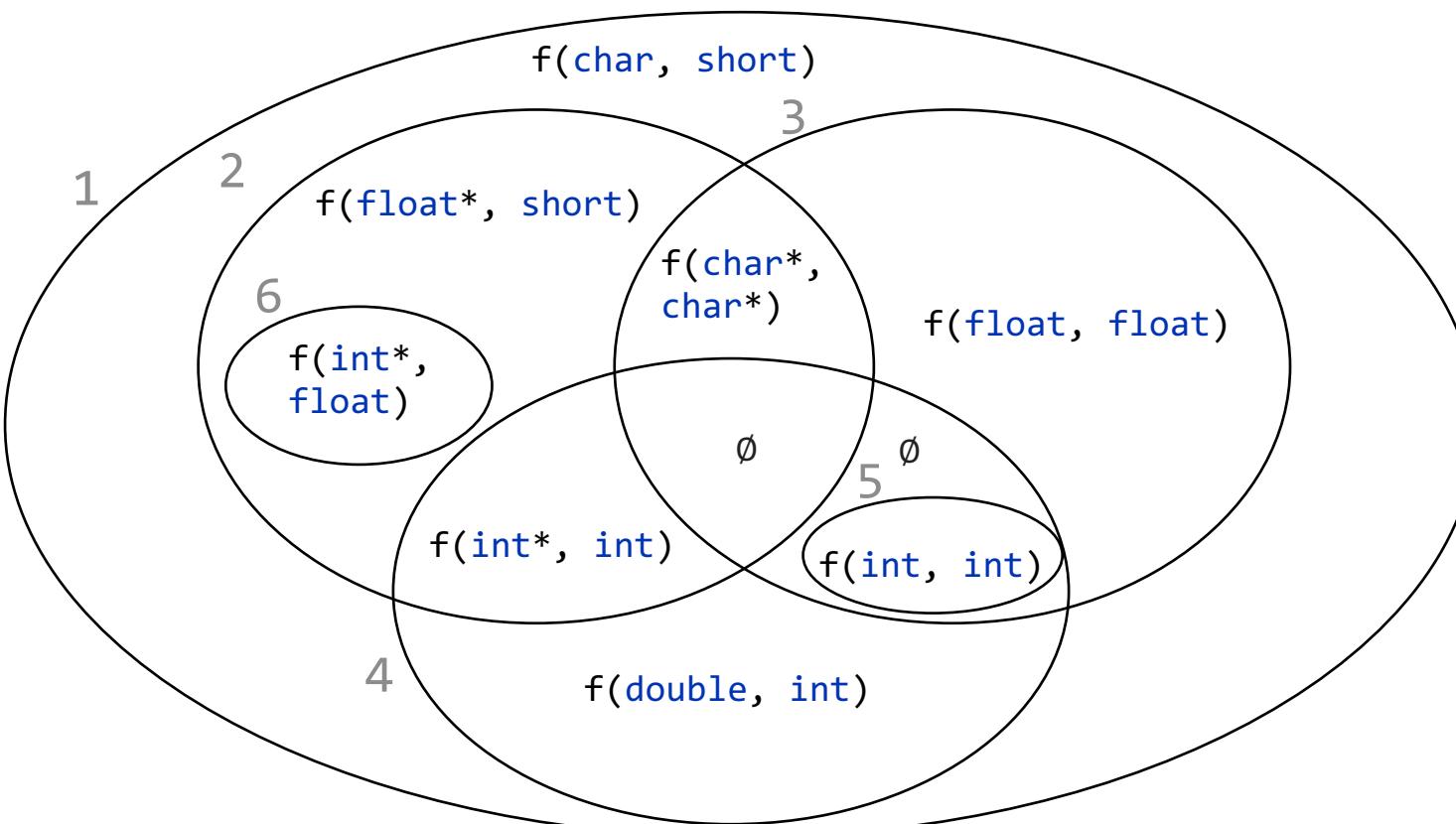


- Quand `f` a plusieurs paramètres, des ensembles non-inclus l'un dans l'autre peuvent avoir une intersection non vide
- Ces intersections seront des lieux d'ambiguïté
- Par exemple, `f(char*, char*)`
 - peut appeler 1, 2 ou 3
 - 2 et 3 sont plus spécialisées que 1, mais il n'y a pas d'ordre entre elles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



Avec des fonctions non génériques

```
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



Algorithme général de résolution

1. Etablir 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

```

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

R1 := 1^{er} paramètre = `int`

- 1 : $\langle T, U \rangle = \langle \text{int}, ? \rangle$
- 2 : pas de déduction possible
- 3 : $\langle T \rangle = \langle \text{int} \rangle$
- 4 : $\langle T \rangle = \langle \text{int} \rangle$
- 5 : type exact
- 6 : pas de conversion

R2 := 2^{ème} paramètre = `int`

- $\langle T, U \rangle = \langle ?, \text{int} \rangle$
- $\langle T, U \rangle = \langle ?, \text{int} \rangle$
- $\langle T \rangle = \langle \text{int} \rangle$
- $\langle T \rangle = \langle ? \rangle$, type exact
- type exact
- ajustement `int -> float`

R1 ∩ R2

- $\langle T, U \rangle = \langle \text{int}, \text{int} \rangle$
- pas appellable
- $\langle T \rangle = \langle \text{int} \rangle$
- $\langle T \rangle = \langle \text{int} \rangle$
- appelable
- pas appellable



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

```
// Déductions :  
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 appellable  
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
```

Etape 2.1 : résolution au sens du chapitre 4

R1 := 1^{er} paramètre = `int`

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

R2 := 2^{ème} paramètre = `int`

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

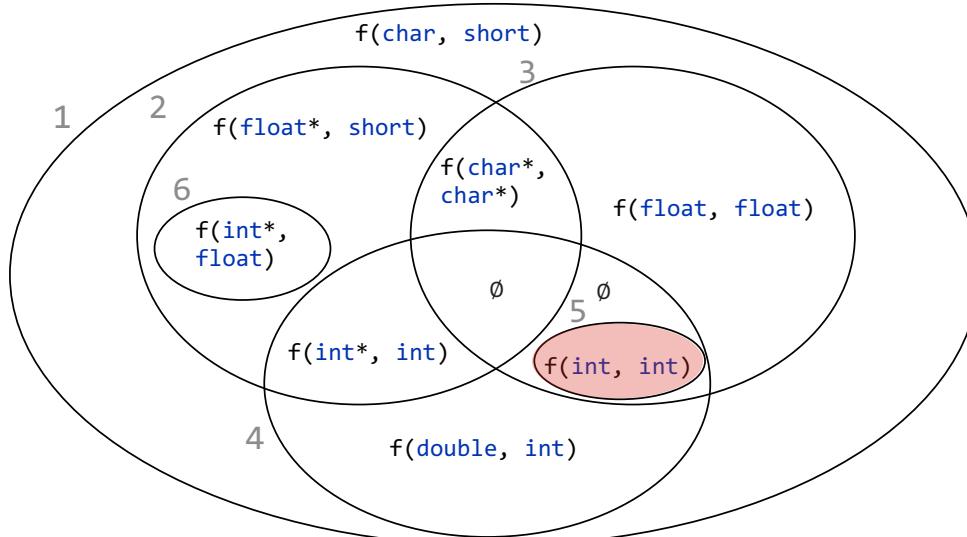
R1 = { 1, 3, 4, 5 }
R2 = { 1, 3, 4, 5 }
R1 ∩ R2 = { 1, 3, 4, 5 }



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

```
// Déductions :  
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 appellable  
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
```

Etape 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

```

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

R1 := 1^{er} paramètre = `char`

- 1 : $\langle T, U \rangle = \langle \text{char}, ? \rangle$
- 2 : pas de déduction possible
- 3 : $\langle T \rangle = \langle \text{char} \rangle$
- 4 : $\langle T \rangle = \langle \text{char} \rangle$
- 5 : promotion `char` -> `int`
- 6 : pas de conversion

R2 := 2^{ème} paramètre = `int`

- $\langle T, U \rangle = \langle ?, \text{int} \rangle$
- $\langle T, U \rangle = \langle ?, \text{int} \rangle$
- $\langle T \rangle = \langle \text{int} \rangle$
- $\langle T \rangle = \langle ? \rangle$, type exact
- type exact
- ajustement `float` -> `int`

R1 ∩ R2

- $\langle T, U \rangle = \langle \text{char}, \text{int} \rangle$
- pas appellable
- pas appellable
- $\langle T \rangle = \langle \text{char} \rangle$
- appelable
- pas appellable



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

```

// Déductions :
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 appellable
template <typename T> void f(T, T) {...} // 3 pas appellable
template <typename T> void f(T, int) {...} // 4 void f<char>(char, int);
void f(int, int) {...} // 5
void f(int*, float) {...} // 6

```

Etape 2.1 : résolution au sens du chapitre 4

R1 := 1^{er} paramètre = `char`

1 : **type exact**
2 : N/A
3 : N/A
4 : **type exact**
5 : promotion char -> int
6 : N/A

R2 := 2^{ème} paramètre = `int`

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

R1 = { 1, 4 }

R2 = { 1, 4, 5 }

R1 ∩ R2 = { 1, 4 }



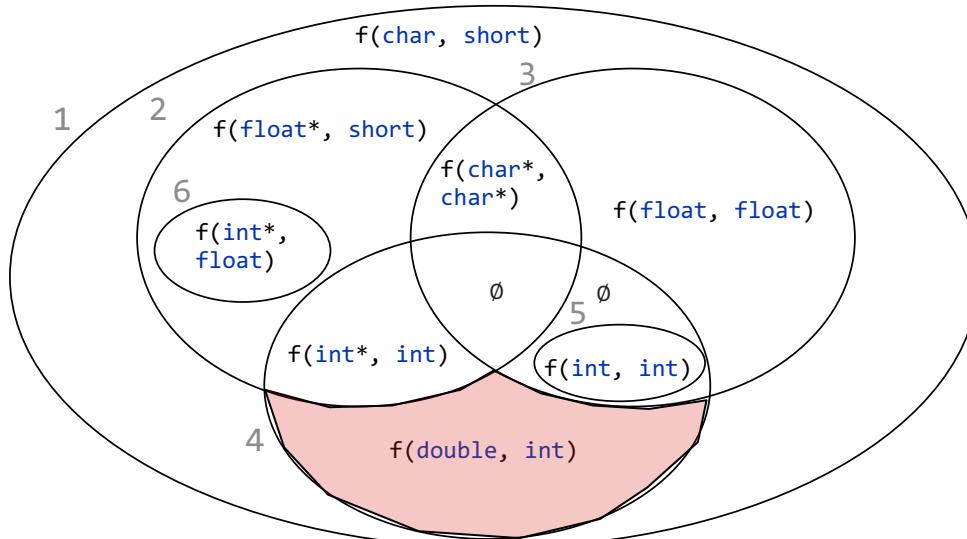
Exemple 2 : char c; int i; f(c, i);

```

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

```

Etape 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
```

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

R1 := 1^{er} paramètre = `int*`

- 1 : `<T,U> = <int*,?>`
- 2 : `<T,U> = <int,?>`
- 3 : `<T> = <int*>`
- 4 : `<T> = <int*>`
- 5 : pas de conversion
- 6 : type exact

R2 := 2^{ème} paramètre = `int*`

- `<T,U> = <?,int*>`
- `<T,U> = <?,int*>`
- `<T> = <int*>`
- pas de conversion
- pas de conversion
- pas de conversion

R1 ∩ R2

- `<T,U> = <int*,int*>`
- `<T,U> = <int,int*>`
- `<T> = <int*>`
- pas appellable
- pas appellable
- pas appellable



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

```
// Déductions :  
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
```

Etape 2.1 : résolution au sens du chapitre 4

R1 := 1^{er} paramètre = `int*`

R2 := 2^{ème} 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 ∩ R2 = { 1, 2, 3 }



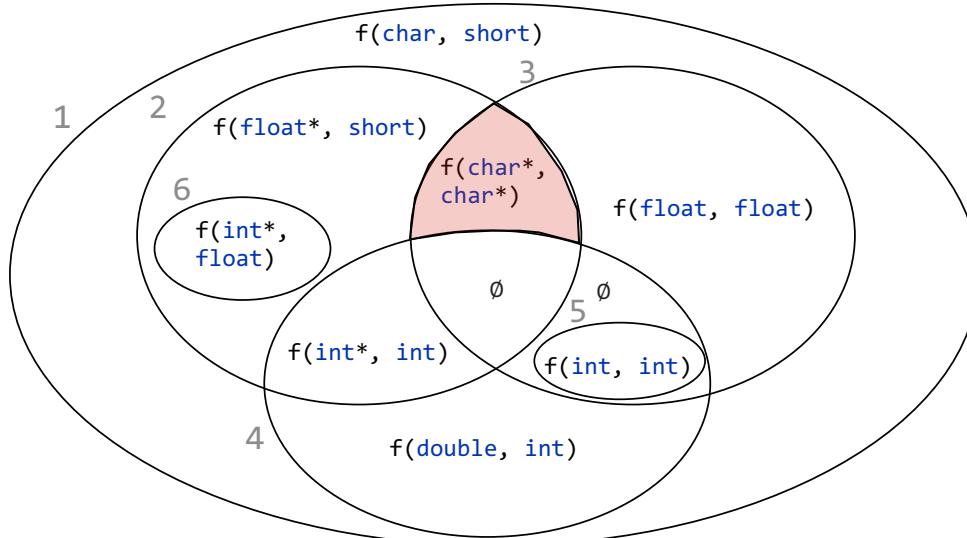
Exemple 3 : int *p, *q; f(p, q);

```

// Déductions :
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

```

Etape 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 spécialiser une fonction générique pour un argument générique donné

- en utilisant le mot clé `template<>` avec `<>` obligatoire
- suivi de la fonction où tous les arguments génériques sont spécifiés

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

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

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



- Dans l'exemple précédent, pour spécialiser pour `T=int`

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

- On aurait pu écrire indifféremment

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

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

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

- Le type `T` étant déductible de l'argument de la fonction



Spécialisation - avec surcharge

- Lors de la déduction des arguments d'une spécialisation, les mêmes règles que pour les instanciations s'appliquent en cas de surcharge

```
template <typename T> void f(T t) { cout << 1; }
template <typename T> void f(T* t) { cout << 2; } // surcharge
template<> void f<>(int* t) { cout << 3; } // spécialisation
```

- La fonction 3 spécialise la surcharge 2, le type `int*` pouvant appeler 1 avec `T=int*` ou 2 avec `T=int` mais la surcharge 2 étant plus spécialisée
- Attention, l'ordre importe. Ci-dessous, la fonction 3 spécialise la surcharge 1, la surcharge 2 n'étant pas encore déclarée lors de la définition de 3.

```
template <typename T> void f(T t) { cout << 1; }
template<> void f<>(int* t) { cout << 3; } // spécialisation
template <typename T> void f(T* t) { cout << 2; } // surcharge
```



Spécialisation - avec surcharge (2)

- On peut expliciter quelle surcharge est spécialisée en n'utilisant pas la déduction d'argument générique mais en les explicitant

```
template <typename T> void f(T t) { cout << 1; }
template <typename T> void f(T* t) { cout << 2; } // surcharge
template<> void f<int*>(int* t) { cout << 3; } // spécialisation
template<> void f<int>(int* t) { cout << 4; } // spécialisation
```

- La fonction 3 spécialise la fonction 1
- La fonction 4 spécialise la surcharge 2



Spécialisation – déduction à l'appel

- Savoir quelle surcharge est spécialisée est important, car seules les surcharges participent à l'algorithme de résolution, pas les spécialisations

```
template <typename T> void f(T t) { cout << 1; }
template <typename T> void f(T* t) { cout << 2; } // surcharge
template<> void f<int*>(int* t) { cout << 3; } // spécialisation de 1

int main() {
    f((int*)(0)); // affiche 2
}
```

- L'appel avec un paramètre effectif de type `int*` choisit entre les surcharges 1 et 2, et donc préfère `T*` à `T`.
- La fonction 3 spécialise 1, et n'est donc pas appelée



Argument générique énumérable

- En plus des `typename`, on peut utiliser des arguments génériques énumérables, i.e. de type entier ou enum, dont la valeur doit être connue à la compilation

```
template <int n, typename T>
T multiplier(const T& t) {
    return (T)n * t;
}

int main() {
    cout << multiplier<3, double>(2.5); // affiche 7.5
    cout << multiplier<5>(2.5); // par déduction, affiche 12.5
    cout << multiplier< >(2.5); // ne compile pas. n non déductible
}
```



Argument générique énumérable (2)

- Cela permet notamment d'écrire des fonctions traitant des `std::array` de taille quelconque.

```
template <typename T, size_t n>
void afficher(const array<T,n> & t) {
    cout << '[';
    for (size_t i = 0; i < n; ++i) {
        if (i) cout << ", ";
        cout << t[i];
    }
    cout << ']';
}

int main() {
    array<int, 5> a{1, 2, 3};
    afficher(a);
}
```

```
[1, 2, 3, 0, 0]
```



Argument générique pointeur ou référence

- Enfin, on peut aussi utiliser un pointeur ou une référence à un objet alloué statiquement ou à une fonction comme argument générique.

```
template <ostream& out>
void afficher(const vector<int> &v) {
    for(int e : v)
        out << e << ' ';
}

int main() {
    vector<int> v{1,2,3};
    afficher<cout>(v);
    afficher<clog>(v);
}
```



Fonctions en paramètre

- Il est possible de passer des fonctions en paramètres d'autre fonctions, de sorte que le code suivant compile et affiche 1.2.3.5.7.

```
void afficher(int i) { cout << i << '.'; }

int main() {
    vector v{1, 2, 3, 5, 7};
    pour_tout(v, afficher);
}
```

```
void pour_tout(const vector<int> & v,
               void(*fct)(int) )
{
    for(int t : v)
        fct(t);
}
```

- La syntaxe avec pointeur de fonction est complexe et restrictive, seule une fonction ayant exactement les même paramètres et type de retour étant acceptée.
- Rendre le type de fct générique simplifie la syntaxe et permet plus de souplesse lors de l'appel (type de retour quelconque, paramètres avec conversion possible, ...)

```
template <typename Fn>
void pour_tout(const vector<int> & v,
               Fn fct)
{
    for(int t : v)
        fct(t);
}
```



Fonctions en paramètre (2)

- Attention, si la fonction à passer en paramètre est elle-même générique,
 - c'est une instantiation de cette fonction générique qu'il faut passer en paramètre, i.e.
afficher<int>
 - pas le template non instancié, i.e pas afficher

```
template <typename T>
void afficher(T t) { cout << t << ' ' ; }

template<typename Fn>
void pour_tout(const vector<int> & v, Fn fct) {
    for(int t : v)
        fct(t);
}

int main() {
    vector v{1, 2, 3, 5, 7};
    pour_tout(v, afficher<int>);
}
```

```
// par contre, le code suivant ne compile pas
pour_tout(v, afficher);
```



La **définition d'une fonction générique** ne définit pas réellement une fonction, mais un moule → deux possibilités en termes 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)

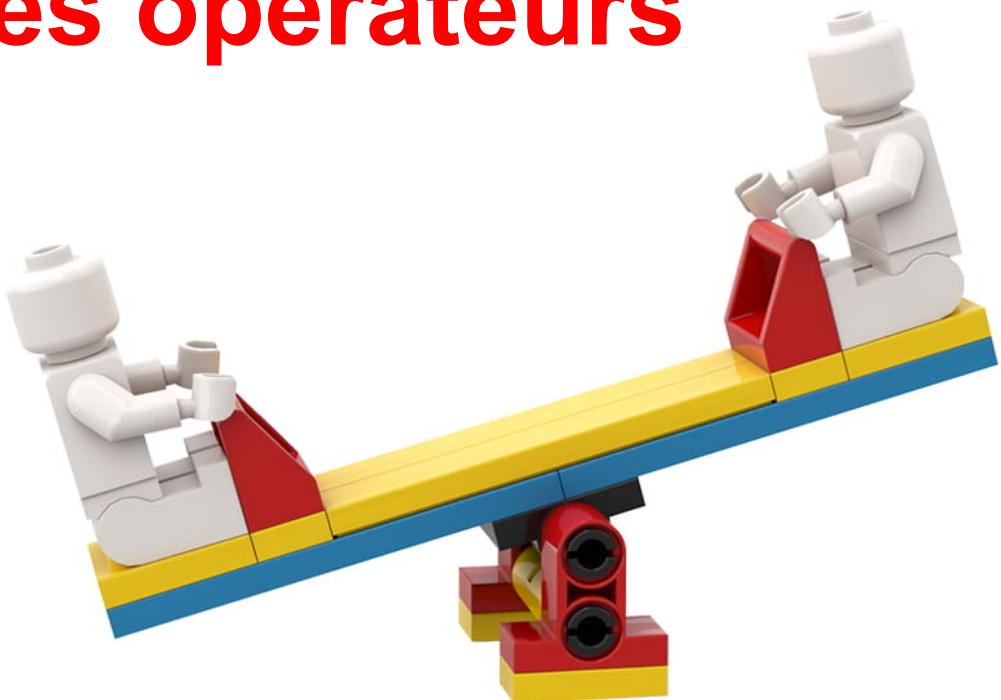


- L'instanciation *implicite* des *templates* rend la **tâche du compilateur complexe** dans un cadre de compilation séparée
- Chaque instantiation implicite dans un fichier .cpp entraîne la **génération de code** dans le fichier objet correspondant
 - ce code doit être nettoyé par l'éditeur de liens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

3. Surcharge des opérateurs





Surcharge opérateur

- Comment réaliser une **opération de comparaison** entre deux dates ?

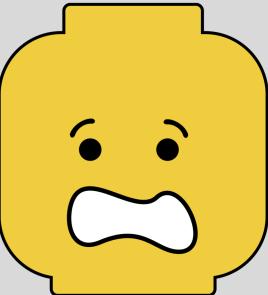
```
struct Date {  
    int jour;  
    int mois;  
    int annee;  
};
```

```
Date iPhone1 {9, 1, 2007};  
Date appleWatch1 {9, 9, 2014};
```

- Pour l'instant avec une fonction implémentant la comparaison souhaitée

```
bool avant(const Date& lhs, const Date& rhs) {  
    if (lhs.annee != rhs.annee)  
        return lhs.annee < rhs.annee;  
    if (lhs.mois != rhs.mois)  
        return lhs.mois < rhs.mois;  
    return lhs.jour < rhs.jour;  
}
```

```
if (avant(iPhone1, appleWatch1)) {  
    ...  
}
```





Surcharge opérateur

- Pour les types simples, nous utilisons souvent des opérateurs
=, +, -, *, ++, <, ==, <<, ...
- Pourrait-on aussi les utiliser ?
Oui, on peut **surcharger les opérateurs** suivants :

+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	
~	&=	^=	=	&&		%=	[]	(())	,	->*	->	new
delete	new[]	delete[]										

- Note 1 : on ne peut pas changer leur priorité ni le nombre d'arguments, et on ne peut pas définir de nouveaux opérateurs
- Note 2 : l'opérateur d'affectation (=) est défini par défaut pour les classes



Surcharge opérateur

- Pour surcharger un opérateur, il faut définir une fonction membre `operator@` où @ est le symbole de l'opérateur que l'on veut définir.
La syntaxe générale est :

```
typeRetour operator @ (paramètres) { /*... corps ...*/ }
```

- Les espaces autour du symbole sont facultatifs
- Les paramètres et le type de retour dépendent de l'opérateur que l'on veut surcharger
- On doit considérer quels paramètres sont modifiés ou pas (mot clé const) et penser au passage par référence& (si la copie de l'objet passé en paramètre est coûteuse)



Surcharge opérateur

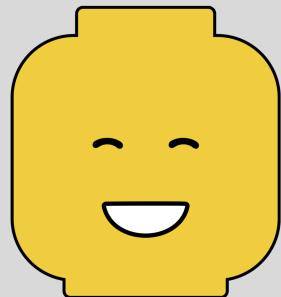
- En surchargeant l'opérateur <, la lecture est beaucoup plus naturelle

```
struct Date {  
    int jour;  
    int mois;  
    int annee;  
};
```

```
Date iPhone1 {9, 1, 2007};  
Date appleWatch1 {9, 9, 2014};
```

```
bool operator< (const Date& lhs, const Date& rhs) {  
    if (lhs.annee != rhs.annee)  
        return lhs.annee < rhs.annee;  
    if (lhs.mois != rhs.mois)  
        return lhs.mois < rhs.mois;  
    return lhs.jour < rhs.jour;  
}
```

```
if (iPhone1 < appleWatch1) {  
    ...  
}
```





Surcharge opérateur

Rien n'oblige que les deux opérandes d'un opérateur soient du même type.

Exemple : ajout de n jours à une date

```
Date operator+ (Date d, int n) {  
    // calcul date + n  
    return d;  
}
```

NB : Il faudra certainement faire une copie de la date reçue en paramètre pour la valeur à retourner. Le paramètre *d* est volontairement passé par copie.

Les paramètres avec valeur par défaut ne sont pas possibles

```
Date operator+ (Date d, int n=1);
```

error: parameter of overloaded 'operator+'
cannot have a default argument



Surcharge opérateur

- On peut alors écrire

```
Date d {1, 2, 2023};  
Date d2 = d + 7; // 8.2.2023
```

- Par contre, si on change l'ordre, le code ne compile pas

```
Date d {1, 2, 2023};  
Date d2 = 7 + d;
```

error: invalid operands to binary
expression ('int' and 'Date')

- En effet, il faut aussi surcharger **operator+** pour **int**, **Date**
et simplement faire un appel à l'autre fonction en **croisant les paramètres**

```
Date operator+ (int n, Date d) {  
    return d + n; // operator+ (Date, int)  
}
```



Surcharge opérateur <<

- Une application fréquente est la surcharge de l' opérateur de flux <<

```
ostream& operator<< (ostream& os, const Date& d) {  
    os << d.jour << '.' << d.mois << '.' << d.annee;  
    return os;  
}
```

```
cout << "First iPhone : " << iPhone1 << endl;  
// First iPhone : 9.1.2007
```

- Les deux paramètres sont
 - une **référence au flux de sortie** (cout, cerr) dans lequel écrire
 - une **référence constante à l'objet** à afficher
- On retourne la référence au même flux



Surcharge opérateur >>

- Ceci peut également être utilisé pour le **flux d'entrée >>**

```
istream& operator>> (istream& is, Date& d) {  
    is >> d.jour >> d.mois >> d.annee;  
    return is;  
}
```

```
cout << "saisie : ";  
Date date;  
cin >> date;
```

- Le flux peut volontairement être mis en erreur si la date saisie n'est pas possible



Surcharge opérateur – affect. composée

- Quand on définit un opérateur **binaire** tel que +, -, *, ...
... on devrait aussi définir l'opérateur d'**affectation composée** correspondant
+=, -=, *=, ...
- L'opérateur **d'affectation** retourne typiquement **une référence**
- alors que l'opérateur **binaire** retourne typiquement l'objet **par valeur**

```
Date& operator+=(Date& d, int n) {  
    // calcul date + n  
    return d;  
}
```

```
Date operator+ (Date d, int n) {  
    return d += n;  
}
```

```
Date operator+ (int n, Date d) {  
    return d + n;  
}
```



Surcharge opérateur – forme canonique

- Surcharge de `+` grâce à `+=`
 - pour ne pas dupliquer le code (surtout s'il est complexe)
 - ce serait faux (et inutile) de passer `d` par référence, car la somme doit être un nouvel objet
- L'opérateur `+=` retourner une référence
 - `(ref += 1) += 2` // possible
 - `X& n'est pas const` car il doit être modifié (affecté)
 - permet certaines optimisations ...

```
Date& operator+=(Date& d, int n) {  
    // calcul date + n  
    return d;  
}
```

```
Date operator+(Date d, int n) {  
    return d += n;  
}
```

```
Date operator+(int n, Date d) {  
    return d += n;  
}
```



Surcharge opérateur – op. relationnels

Pour les opérateurs de comparaison et d'égalité, on met typiquement en œuvre `operator<` et `operator==`, et on implémente les autres grâce à ces deux.

```
bool operator< (const X& lhs, const X& rhs) /* comparaison < à écrire ici */  
bool operator> (const X& lhs, const X& rhs) {return rhs < lhs;}  
bool operator<=(const X& lhs, const X& rhs) {return !(rhs < lhs);}  
bool operator>=(const X& lhs, const X& rhs) {return !(lhs < rhs);}  
  
bool operator==(const X& lhs, const X& rhs) /* comparaison == à écrire ici */  
bool operator!=(const X& lhs, const X& rhs) {return !(lhs == rhs);}
```



Surcharge opérateur – operator++

- Pour surcharger l'opérateur `++` (ou `--`), il faut écrire deux fonctions. L'une pour l'opérateur préfixe, l'autre pour le postfixe. Comment les distinguer ?
- **L'opérateur préfixe** ne prend pas de paramètre et retourne une référence vers l'objet lui-même
- **L'opérateur postfixe** prend formellement un paramètre entier (dont la valeur n'est pas utilisée) et retourne une copie de l'objet avant incrémentation

```
Date& operator++ (Date& d) {  
    return d += 1;  
}
```

```
Date operator++ (Date& d, int) {  
    Date temp = d;  
    ++d;  
    return temp;  
}
```