# La surcharge de fonctions et résolution des noms

## Plusieurs fonctions avec le même nom

Le nom d'une fonction est le premier indicateur du rôle d'une fonction pour les utilisateurs de cette fonction. Il est donc important de donner un nom qui exprime le mieux ce rôle. Mais comment faire si vous souhaitez avoir plusieurs fonctions qui exécute la même tâche, mais sur des types différents ?

Par exemple, si vous souhaitez créer une fonction `add` pour additionner des entiers ou des réels. Une première solution est de donner des noms différents aux fonctions.

```
int add_int(int lhs, int rhs);
double add_double(double lhs, double rhs);
```

Note : Pour les opérateurs binaires, il est classique d'utiliser les noms `lhs` pour *left hand side* (côté gauche) et *right hand side* (côté droit).

C'est une approche possible, mais si vous pensez au nombre de types que vous avez vu jusqu'à maintenant, vous comprendrez facilement que cela va vite devenir compliqué. (Mais pas impossible, puisque c'est ce que l'on fait dans certains langages de programmation, comme le C).

## La surcharge de fonctions

En fait, il n'est absolument pas nécessaire de donner un nom différent à ces deux fonctions. Une fonction sera identifiée par le compilateur grâce à sa signature, c'est à dire son nom et la liste des types de ses

paramètres.

Par exemple, pour la fonction suivante :

```
int f(int i, double x, std::string s)
```

Sa signature est :

```
f(int, double, std::string)
```

La surcharge de fonction consiste à definir plusieurs fonctions avec le meme nom, mais des signatures differentes. Lors de l'appel de ces fonctions (en utilisant le nom de la fonction donc), le compilateur cherchera la signature qui s'adaptera le mieux aux types des arguments. (Vous voyez ici, encore, l'importance des types en C++).

Pour la fonction `add` précédente :

```
int add(int i, int j);
double add(double x, double y);

int add(123, 456);    // appel de la première version de add
int add(1.23, 4.56);  // appel de la seconde version de add
```

**Les polymorphismes**

La surcharge de fonction (*overloading* en anglais) est une forme de polymorphisme, le polymorphisme ad-hoc.

Le polymorphisme (dont l'étymologie signifie “qui peut prendre plusieurs formes”) est un terme général, qui désigne (en programmation) quelque chose (fonction, classe, etc) qui peut avoir plusieurs comportement différents, selon le contexte.

Il existe plusieurs formes de polymorphisme, dont les templates citées juste avant (polymorphisme paramétrique), ou l'héritage de classes (polymorphisme d'inclusion), qui sera vu dans la partie sur la programmation orientée objet.

## Fonctions génériques

Une solution alternative pour cette problématique est d'utiliser une fonction générique. Une fonction générique est simplement une fonction qui n'est pas écrite pour un type de paramètre en particulier, mais utilise l'inférence de type pour déterminer les types des parametres.

Par exemple, pour la fonction `add`, il serait possible d'écrire :

```
template<typename T>
T add(T lsh, T rhs);
```

Vous verrez dans le prochain chapitre comment creer des fonctions génériques, ne vous préoccupez pas trop de la syntaxe pour le moment. Sachez simplement que le type `T` dans les paramètres de la fonction sera remplacé par un type concret (`int`, `double`, etc.) lors de l'appel de la fonction.

Le point important est que pour la surcharge de fonction, vous devez écrire une fonction pour chaque type que la fonction pourra accepter. Et pour les fonctions générique, il est nécessaire d'écrire qu'une seule fois la fonction.

(La surcharge de fonctions et les fonctions génériques sont des concepts indépendants, ils peuvent être utiliser ensemble pour creer des surcharges de fonctions génériques).

## D'autres exemple de surcharge de fonctions

Un autre exemple de surcharge de fonction, que vous connaissez bien... sans le savoir. Lorsque vous écrivez :

```
std::cout << i << std::endl;
```

En fait, l'opérateur `«` est une fonction qui prend deux paramètres : le flux

de sortie (`std::cout` dans ce code) et une valeur (`i` par exemple dans ce code). Le code précédent peut donc se traduire :

```
operator<<(operator<<(std::cout, i), std::endl);
```

Ou encore, en faisant apparaître l'expression intermédiaire :

```
std::cout = operator<<(std::cout, i);          // execute
"std::cout << i"
std::cout = operator<<(std::cout, std::endl);  // execute
"std::cout << std::endl"
```

L'opérateur `«` est en fait une surcharge de fonction, qui accepte de nombreux types comme second paramètre. Lorsque le compilateur rencontre l'expression `std::cout « i`, il determine quel est le type de `i`, puis appelle l'opérateur `«` adéquate (sans conversion si possible, avec la conversion la plus simple sinon).

Et c'est la même chose pour les autres opérateurs que vous connaissez ( `+`, `*`, etc.)

```
c = a + b;
// est equivalent a
c = operateur+(a, b);
```

L'étude des opérateurs et leur surcharge est suffisamment important pour être détaillé dans un chapitre dédié, dans la partie sur la programmation orientée objet.

## Cas particulier des references

Vous avez vu dans le chapitre précédent qu'il existe plusieurs types de passage de valeurs dans une fonction : par valeur ou par références (constante ou non, *lvalue* ou *rvalue*).

Le point important a retenir est quelle type de paramètre accepte quel type d'argument. Cela était résumé dans le tableau suivant :

| Passage | lvalue | rvalue |
|---|---|---|
| Par valeur | oui | oui |
| Référence constante | oui | oui |
| Référence | oui | **non** |
| Rvalue-reference | **non** | oui |

Il est possible de surcharger des fonctions pour écrire du code spécifique, selon si la fonction est appellee avec une *lvalue* (une variable) ou une *rvalue* (un temporaire).

Mais pour éviter les ambiguïtés lors de l'appel de fonction, il ne faut pas écrire deux fonctions qui acceptent le même type de valeurs. Par exemple, si vous écrivez :

```
void f(int i);    // passage par valeur
void f(int && i); // passage par rvalue-reference
```

Si vous appelez cette fonction avec une *lvalue*, il n'y aura pas d'ambiguïté (seul le passage par valeur sera valide). Si vous appelez avec une *rvalue*, il y a ambiguïté (les deux versions de la fonction acceptent les *rvalues*).

Il y a donc que trois approches possibles :

1. Si vous ne voulez pas écrire de code spécifique *lvalue* vs *rvalue* et que la copie n'est pas un problème (par exemple pour les types fondamentaux comme `int`, `double`, etc.), alors vous utilisez le passage par valeur.

```
#include <iostream>

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

int main() {
    int i { 123 };
    f(i);
    f(456);
```

```
}
```

affiche :

```
f(int i): 123
f(int i): 456
```

2. Si vous ne voulez pas écrire de code spécifique *lvalue* vs *rvalue* et que la copie est un problème (par exemple pour les classes comme `std::string`, `std::vector`, etc.), alors vous utilisez le passage par référence constante.

```
#include <iostream>
#include <string>

void f(std::string const& s) {
    std::cout << "f(std::string const& s): " << s << std::
endl;
}

int main() {
    std::string s { "hello" };
    f(s);
    f("world");
}
```

affiche :

```
f(std::string const& s): hello
f(std::string const& s): world
```

3. Si vous voulez écrire de code spécifique *lvalue* vs *rvalue*, alors vous utilisez le passage par référence non constante (c'est a dire que vous écrivez deux fonctions surchargées, qui acceptent une *lvalue-reference* et une *rvalue-reference*).

```
#include <iostream>

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

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

int main() {
    int i { 123 };
    f(i);
    f(456);
}
```

affiche :

```
f(int & i): 123
f(int && i): 456
```

Ce type de surcharge de fonction sera particulièrement intéressant lorsque vous concevrez vos propres classes, puisque cela permet d'optimiser la gestion des données internes, selon le type de valeurs utilisées. C'est ce qui est fait dans la bibliothèque standard. Par exemple, pour `std::vector`, vous pouvez voir dans la documentation ( std::vector::vector) :

```
vector( const vector& other );            (5)
vector( vector&& other )                  (6)
```

## Résolution des noms de fonctions

Plusieurs fonctions avec le même nom pose le problème de déterminer quelle fonction sera effectivement appelée lors d'un appel de fonction. Les règles qui définissent comment le compilateur détermine cela est appelé “la résolution des noms” (name lookup en anglais).

Par exemple, avec le code du “hello world” :

```
#include <iostream>

int main() {
    std::cout << "hello, world!" << std::endl;
```

```
}
```

Lorsque le compilateur analyse ce code, il va trouver les identifiants suivants :

- `int`
- `main`
- `std`
- `cout`
- `endl`

La résolution des noms est donc le processus qui permet au compilateur de déterminer ce que signifie chacun de ces identifiants.

Notez que ce chapitre se limite aux cas simple de resolution de noms. Il existe des regles supplementaires pour les fonctions génériques ( *template*), les classes. Ces règles seront vue dans les cours correspondants.

## Analyse sequentielle

Vous avez vu au début de ce cours qu'en programmation impérative, les instructions étaient exécutées les unes après les autres (dans Le programme "hello world"). Le compilateur fonctionne de la même façon : il lit le code ligne par ligne, de façon purement séquentielle. Lorsqu'il lit une ligne, il utilise les informations qu'il a appris en lisant les lignes précédentes.

C'est pour cette raison que vous avez vu qu'il fallait definir une fonction avant de l'utiliser.

```
void f() {}

int main() {
    f();  // ok, le compilateur connaît déjà f
    g();  // erreur, le compilateur ne connaît pas encore g
}
```

```
void g() {}
```

Une exception à cette règle : le compilateur connaît un certain nombre d'identifiants par défaut, définis dans la norme C++. Ces identifiants sont réservés, il vous est interdit de les utiliser. Ce sont les mots-clés du langage. Vous connaissez deja :

- `auto` : inférence de type ;
- `bool` : type booléen ;
- `char` : type d'entier de taille la plus petite (`sizeof(char) == 1`), representant un caractere ;
- `class` : pour definir une structure de donnees (classe) ;
- `const` : constant (vous avez vu ce mot-clé pour les variables, mais il sera utilisé aussi pour les fonctions membres) ;
- `constexpr` : expression constante (évaluée lors de la compilation si possible) ;
- `decltype` : inférence de type ;
- `double` : type de nombre à virgule flottante, généralement sur 64 bits ;
- `enum` : enumeration ;
- `false` : valeur booleenne “faux” ;
- `float` : type de nombre à virgule flottante, généralement sur 32 bits ;
- `int` : type d'entier, intermédiaire entre `short int` et `long int`, généralement sur 32 bits ;
- `long` : modificateur de type, permet d'utiliser des entiers (`int`) et réels (`double`) de plus grande taille ;
- `operator` : pour definir un opérateur (fonction particulière, comme `+`, `*`, `«`, etc.) ;
- `return` : retourne une valeur et termine une fonction ;
- `short` : modificateur de type, permet d'utiliser des entiers (`int`) de plus petite taille ;
- `signed` : modificateur de type, pour creer une entier signé ;
- `sizeof` : opérateur permettant de connaître la taille en mémoire

d'un type ou d'une variable ;
- `struct` : pour definir une structure de données ;
- `template` : pour definir une fonction ou une classe générique ;
- `true` : valeur booléenne “vrai” ;
- `typedef` : ancienne syntaxe pour definir un alias de type ;
- `unsigned` : modificateur de type, pour creer une entier non signe ;
- `using` : permet de creer un alias de type ou de specifier un espace de noms ;
- `void` : indique qu'une fonction ne retourne pas de valeur.

Certain de ces mots-clés ont des utilisations que vous n'avez pas encore vues et il existe d'autres mots-clés que vous verrez par la suite. La liste complete des mots-cles est indiquee dans la documentation : C++ keywords.

## Directive de préprocesseur

Dans le code du programme “hello world”, la première instruction rencontrée par le compilateur est la directive de préprocesseur `#include`. Il existe plusieurs directives de compilation, elles seront expliquées dans un chapitre dédié.

Une directive commence toujours par un dièse `#`. La directive `#include` permet d'utiliser les fonctionnalités déclarées dans un fichier d'en-tete (de la bibliotheque standard pour le moment, mais vous verrez plus tard qu'il est possible de creer ses propres fichiers d'en-tête).

Il est possible de mettre la directive `#include` a n'importe quel endroit (en dehors des fonctions), mais la règle séquentielle s'applique : vous ne pouvez pas utiliser une fonctionnalité d'un fichier d'en-tete avant d'avoir inclue ce fichier d'en-tete.

```
void f() {
    std::cout << "hello" << std::endl;  // erreur, std::cout
et std::endl ne
```

```
                                        // sont pas encore
connus
}

#include <iostream>

void g() {
    std::cout << "world" << std::endl;  // ok
}
```

Pour faciliter la lecture et éviter les erreurs, il est classique de placer les directives `#include` au debut du code. Cette règle sera utilisée dans ce cours.

## Nom qualifié et non-qualifié

Dans le code du programme “hello world”, vous avez vu comment le compilateur va interpreter les premiers identifiants : la directive `#include` et le mot-cle `int`. Il va également comprendre que l'identifiant `main` est la définition d'une nouvelle fonction et qu'il ne connaît pas encore ce nom, il va donc l'ajouter a sa liste des noms connus.

Le compilateur arrive ensuite à l'identifiant `std`. Il sait que les doubles deux-points `::` correspondent à l'opérateur de portée, qui est utilisé avec les espaces de noms, les classes ou les énumérations par exemple. L'espace de noms `std` est défini dans le fichier d'en-tête `iostream`, donc le compilateur connaît déjà cet identifiant quand il arrive a l'instruction `std::cout`.

Un identifiant précédé d'une portée (une classe, un espace de noms, énumération, etc.) est appelee “nom qualifié” (*qualified name*), sinon il est appelle “nom non-qualifié” (*unqualified name*). Ainsi, `std::cout` et `std::endl` sont des noms qualifiés, alors que `main` et `int` ne sont pas qualifiés.

La portée permet d'utiliser un même identifiant dans plusieurs définitions, à partir du moment ou ces identifiants sont dans des portées différentes. Par exemple, la fonction `size` peut correspondre à des fonctions

différentes, dans ces classes differentes : `std::string::size`, `std::vector::size`, etc. (Il est possible d'avoir plusieurs portées les unes dans les autres).

Dans le chapitre Le programme "hello world", vous avez vu qu'il existe plusieurs syntaxes pour utiliser un espace de noms, en particulier une syntaxe avec `using namespace`. Cette syntaxe permet d'indiquer au compilateur qu'il peut rechercher n'importe quel identifiant aussi dans l'espace de noms `std`.

Par exemple, dans le code suivant, lorsque le compilateur arrivera a l'instruction `cout`, il recherchera dans sa liste des identifiants connus aussi bien `cout` que `std::cout`.

```
#include <iostream>

using namespace std;

int main() {
    cout << "hello world" << endl;
}
```

Dans ce cas, il ne connaît qu'un seul identifiant qui peut correspondre : `std::cout` definie dans `iostream`.

Mais si vous souhaitez ajouter une fonction `cout` :

```
#include <iostream>

using namespace std;

void cout() { std::cout << "hello world" << std::endl; }

int main() {
    cout << "hello world" << endl;
}
```

Dans ce cas, le compilateur produit le message d'erreur suivant pour l'instruction `cout` dans la fonction `main` :

```
main.cpp: In function 'int main()':
main.cpp:9:5: error: reference to 'cout' is ambiguous
     cout << "hello world" << endl;
     ^~~~
main.cpp:5:6: note: candidates are: void cout()
 void cout() { std::cout << "hello world" << std::endl; }
      ^~~~
In file included from main.cpp:1:0:
/usr/local/include/c++/6.1.0/iostream:61:18: note:
std::ostream std::cout
   extern ostream cout;  /// Linked to standard output
                  ^~~~
```

Le compilateur trouve en effet deux définitions pouvant correspondre à `cout` : la fonction `cout` définie dans le code et le flux de sortie `std::cout` définie dans `iostream`. Ces deux identifiants sont valides et le compilateur ne peut déterminer lequel il doit utiliser, il y a ambiguïté ( *reference to* '*cout*' *is ambiguous*).

Notez bien que seul l'instruction `cout` dans la fonction `main` est ambiguë. L'instruction `std::cout` dans la fonction `cout()` est qualifiée, elle n'est pas ambigue.

L'utilisation de `using namespace` réduit l'intérêt des espaces de noms et peut poser des problèmes de conflits dans les noms. L'utilisation de cette syntaxe est généralement limitée aux fichiers sources (vous verrez bientôt la séparation du code entre fichiers d'en-tête et fichiers source).

## Regle de la définition unique et portée

Pour interpréter un code, le compilateur maintient donc une liste des identifiants qu'il connaît et ce qu'ils signifient. En C++, un **déclaration** est une syntaxe qui dit au compilateur qu'un identifiant existe. Une **définition** est une déclaration qui dit au compilateur a quoi correspond un identifiant.

Pour utiliser un identifiant, il faut que :

- celui-ci soit définie et pas simplement déclarée ;
- la définition doit être unique (ODR, *One Definition Rule*, regle de la definition unique).

Par exemple, si vous définissez deux fois la même fonction (même nom et même signature) :

```
void f() {}
void f() {}

int main() {
}
```

affiche le message d'erreur suivant :

```
main.cpp: In function 'void f()':
main.cpp:4:6: error: redefinition of 'void f()'
 void f() {}
      ^
main.cpp:3:6: note: 'void f()' previously defined here
 void f() {}
      ^
```

La règle de la définition unique prend en compte la portee (les blocs de code, les espaces de noms, les classes, etc.). Vous pouvez donc utiliser plusieurs fois le même identifiant, si c'est dans des portées différentes.

```
#include <iostream>

struct MyStruct {
    int i {};  // #1
};

namespace MyNamespace {
    int i {};  // #2
}

void f() {
    int i {};  // #3
}
```

```
int main() {
    int i {};  // #4

    std::cout << i              << std::endl;  // utilise #4
    std::cout << MyStruct::i    << std::endl;  // utilise #1
    std::cout << MyNamespace::i << std::endl;  // utilise #2
    // la variable #3 est une variable locale dans la
fonction f et n'est
    // pas accessible en dehors de cette fonction.
}
```

## Résolution de la surcharge

Lorsque le compilateur rencontre une fonction, il regarde dans la liste des identifiants qu'il connaît pour trouver les fonctions définies avec le même nom. Comme indiqué au début de ce chapitre, il est possible d'avoir plusieurs fonctions qui possedent le meme nom, a partir du moment ou leur signature est différentes (la liste des types de leurs paramètres).

Le compilateur doit donc déterminer exactement quelle fonction appeler. Pour cela, il va regarder la liste des types des arguments utilisés lors de l'appel de la fonction, puis sélectionner la fonction la plus adaptée.

Par exemple, si vous écrivez deux fonctions, l'une qui prend un paramètre de type `int` et l'autre qui prend un paramètre de type `double`. Si vous appelez cette fonction en passant une valeur de type entière, la première version sera automatiquement appellée. Avec une valeur de type réelle, la seconde version sera appellée.

main.cpp

```
#include <iostream>

void f(int) {
    std::cout << "int" << std::endl;
}

void f(double) {
    std::cout << "double" << std::endl;
```

```
}

int main() {
    f(1);   // 1 est une littérale de type int
    f(1.0); // 1.0 est une littérale de type double
}
```

affiche

```
int
double
```

Dans ce code d'exemple, les types des arguments et de paramètres correspondent parfaitement. Il n'y a aucune ambiguïté sur les appels de fonction et le compilateur gère la résolution de la surcharge sans problème.

Mais la résolution de la surcharge peut également fonctionner lorsque les types ne correspondent pas parfaitement. Par exemple, si vous utilisez un argument de type `float`.

```
f(1.0f);
```

affiche :

```
double
```

Dans ce code, l'argument de type `float` est automatiquement promu en type `double`, puis la seconde version de la fonction `f` est appelee.

## Promotion et conversion

Si vous testez la même chose avec un argument de type `long int` :

```
f(1L);
```

Dans ce cas, le compilateur va générer un message d'erreur !

```
main.cpp: In function 'int main()':
main.cpp:12:9: error: call of overloaded 'f(long int)' is
ambiguous
     f(1L);
         ^
main.cpp:3:6: note: candidate: void f(int)
 void f(int) {
      ^
main.cpp:7:6: note: candidate: void f(double)
 void f(double) {
      ^
```

Pourquoi, dans le premier cas, le compilateur arrive à gérer la conversion implicite de `float` vers `double`, mais n'arrive pas a gerer `long int` ?

La raison est qu'il existe plusieurs niveaux de conversion implicite, dans l'ordre de priorité suivant :

- aucune conversion ;
- la promotion ;
- la conversion.

Dans le cas de l'appel de `f(1)`, le compilateur a le choix entre deux fonctions : appeler la fonction `f(int)` sans faire de conversion et appeler `f(double)` en faisant une conversion. La première est prioritaire par rapport à la seconde, il n'y a pas d'ambiguïté. Idem pour l'appel de `f(1.0)`, qui appelle `f(double)` sans conversion.

Dans le cas de l'appel de `f(1.0f)`, le compilateur a le choix entre faire une **promotion** de `float` vers `double` pour appeler `f(double)` ou faire une **conversion** de `float` vers `int` pour appeler `f(int)`. La promotion est prioritaire sur la conversion et `f(double)` est appellee sans ambiguïté.

Pour le dernier cas, l'appel de `f(1L)`, le compilateur a le choix entre deux conversions : `long int` vers `int` et `long int` vers `double`. Ce sont deux conversions, donc avec le même niveau de propriété, le compilateur ne peut pas décider laquelle choisir : il y a ambiguite.

La question est donc de savoir quand une conversion implicite est une

promotion ou non. Le détail des promotions autorisees est donnée dans la documentation : Les promotions numériques.

Pour simplifier, retenez les promotions suivantes :

- la promotions d'un entier plus petit que `int` (`char` ou `short`) en `int` ;
- la promotion de `bool` en `int` ;
- la promotion de `float` en `double`.

La promotion de `bool` en `int` est un reliquat du langage C, qui ne possédait pas de type `bool`. Le type `int` était alors utilisé pour représenter un booléen, avec une valeur nulle pour représenter `false` et une valeur non nulle pour représenter `true`.

Notez aussi que la conversion implicite d'une énumération à portée globale (*unscoped enum*) en entier est également une promotion. Voir Constantes et énumérations.

**Pointeurs et booléens**

Les pointeurs nus sont des types qui permettent de manipuler directement la mémoire en bas niveau. C'est une fonctionnalité avancée du C++, que vous verrez en détail plus tard, mais vous en avez déjà rencontré dans ce cours : les chaines littérales.

```
auto str = "hello, world";  // type "pointeur" : const char*
```

Le problème avec les pointeurs est qu'ils sont convertissable en booléen, ce qui produire des comportements surprenants. Par exemple, si vous écrivez :

```
void foo(bool) { std::cout << "f(bool)" << std::endl; }
void foo(string const&) { std::cout << "f(string)" << std::
endl; }

foo("abc");
```

Ce code ne va pas afficher `f(string)`, mais `f(bool)`.

Si vous ajoutez une fonction `f(const char*)`, celle-ci sera appelée en premier. La raison est que la littérale chaîne est de type `const char*`, les fonctions surchargées ont donc les priorités suivantes, dans l'ordre :

- `f(const char*)`, puisqu'il n'y a pas de conversion nécessaire entre l'argument et le paramètre ;
- `f(bool)`, puisque cela necessite une simple conversion implicite d'un pointeur en `bool` ;
- `f(std::string)`, puisque cela nécessite la construction d'une classe complexe.

Faites attention lorsque vous écrivez une fonction qui prend un paramètre de type `bool`, celle-ci pourra également être appellee avec un argument de type pointeur, en particulier une littérale chaine.

| Chapitre précédent | Sommaire principal | Chapitre suivant |
| --- | --- | --- |