Les paramètres de fonctions

Les variables locales a une fonction ne sont donc pas partageable avec le code qui appelle une fonction, avec d'autres fonctions ou entre plusieurs appels d'une même fonction. Il est donc nécessaire de pouvoir partager des informations entre une fonction et le reste du code. Cela est réalisé via les paramètres de fonction. Pour rappel, la syntaxe générale d'une fonction est la suivante :

```
PARAMETRE_SORTIE NOM_FONCTION (PARAMETRES_ENTREE) {
    INSTRUCTIONS
}
```

Une fonction va donc pouvoir recevoir zéro, un ou plusieurs paramètres en entrées et zéro ou un paramètre en sortie. Pour commencer, vous allez voir les fonctions qui ne prennent que des paramètres en entrée, les paramètres de sortie seront vu ensuite.

Fonction avec un seul paramètre en entrée

La syntaxe pour déclarer un paramètre de fonction est assez simple : il est identique à une déclaration de variable, avec un type et un identifiant (et une valeur par défaut, qui sera vu ensuite. Considérez pour le moment que le paramètre n'a pas de valeur par défaut).

```
TYPE IDENTIFIANT
```

Par exemple, pour une fonction qui prend un entier et une autre qui prend une chaîne de caractères.

```
void f(int i) {
    ...
}

void g(std::string s) {
```

```
    ...
}
```

Pour la fonction `f`, le paramètre est `int i`, qui est déclaré avec un type `int` et une identifiant `i`. Même chose pour `g`, qui contient un paramètre de type `std::string` et qui s'appelle `s`.

En pratique, un paramètre s'utilise de la même façon qu'une variable locale (en particulier, un paramètre suit les mêmes règles concernant la portée et la durée de vie : vous pouvez utiliser un paramètre dès qu'il est déclaré et jusqu'à la fin de la fonction).

Par exemple, pour afficher les valeurs des paramètres des fonctions `f` et `g`.

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

void g(std::string s) {
    std::cout << s << std::endl;
}
```

Le dernier point à voir est comment appeler une fonction qui prend un paramètre. Pour cela, vous devez donner une valeur entre les parenthèses lors de l'appel de la fonction (cette valeur s'appelle un argument dans un appel de fonction). Une valeur peut être une variable (constante ou non), une littérale ou une expression.

Il existe différents modes de passage de paramètres, qui seront vu dans la suite. Chaque mode accepte ou non certaines types de valeurs. Pour le moment, le mode de passage de paramètre vu est le “passage par valeur”, ce qui autorise l'utilisation de n'importe quel type de valeur.

Par exemple, pour appeler une fonction qui prend un paramètre :

```
NOM_FONCTION(VALEUR)
```

Avec le code d'exemple précédent, vous pouvez donc écrire :

```
#include <iostream>

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

void g(std::string s) {
    std::cout << s << std::endl;
}

int main() {
    f(123);        // appel avec une littérale de type
entier
    g("hello");    // appel avec une littérale de type
chaîne

    const int i { 456 };
    f(i);          // appel avec une variable de type entier
    const std::string w { "world "};
    g(w);          // appel avec une variable de type chaîne
}
```

affiche :

```
123
hello
456
world
```

Notez bien que même si dans cet exemple les variables dans la fonction `main` et la fonctions `f` ont le même nom `i`, ce sont bien deux variables différentes (elles ne sont pas dans le même contexte). Comme vous le voyez avec la fonction `g`, il n'y a aucune obligation d'utiliser le même nom entre l'appel de la fonction et le paramètre.

Ce qui est logique et souhaitable. Comme vous pouvez appeler une fonction plusieurs fois avec des valeurs différentes (et donc des variables différentes), cela n'aurait pas de sens que l'identifiant dans le paramètre de fonction soit identique à l'identifiant utilisé pour appeler la fonction.

```
const int i { 123 };
```

```
f(i);  // ok

const int j {456 };
f(j);  // ok
```

Paramètre et argument
Il existe deux termes qui sont proches, mais qui ont un sens légèrement différent : “paramètre” et “argument”. Dans la déclaration de la fonction, vous trouver des “paramètres de fonction”. Dans l'appel de fonction, ce sont des “arguments de fonction”.

```
void f(paramètres...) {
}


int main() {
    f(arguments...);
}
```

Beaucoup de personne confondent les deux notions et cela n'est généralement pas problématique. Mais c'est souvent à ce genre de petits détails que l'on reconnait un développeur qui connait son sujet.

Fonction avec plusieurs paramètres en entrée

Pour déclarer une fonction qui prend plusieurs paramètres, vous devez donner la liste des paramètres séparés par une virgule.

```
TYPE IDENTIFIANT, TYPE IDENTIFIANT, TYPE IDENTIFIANT...
```

Il n'y a pas de limite théorique dans la norme C++ sur le nombre de paramètres que vous pouvez écrire (A vérifier). Par contre, les compilateurs imposent en général une limite. Et de toute façon, une fonction qui prend trop de paramètres deviendra difficilement lisible et c'est généralement le signe qu'il y a un problème d'organisation dans le code.

Par exemple, pour écrire une fonction qui prendre en paramètre un entier et une chaîne :

```
void f(int i, std::string s) {
    std::cout << i << std::endl;
    std::cout << s << std::endl;
}
```

Une fonction qui prend plusieurs paramètres s'appelle en donnant la liste des arguments, séparés aussi par des virgules. Chaque argument est indépendant, vous pouvez mélanger des variables, des constantes, des littérales, des expressions, selon ce qui est autorisé pour chaque paramètre (selon de le mode).

```
f(123, "hello");
```

std::pair, std::tuple, structures de données

Même si un paramètre correspond à un type, rien n'interdit que ce soit un type complexe, comme par exemple une `std::pair`, un `std::tuple`, ou une structure de données. Chacun de ces types peut contenir plusieurs autres types, ce qui permet en pratique de passer en paramètres de fonction autant de types que vous souhaitez.

Il faut donc éviter de passer trop d'information à une fonction et faire en sorte que les paramètres aient une cohérence entre eux. Par exemple, `std::string` est une structure de données complexe, qui contient d'autres information en interne. Mais cela ne pose pas de problème, puisque ce type est manipulé comme un tout cohérent, sans avoir besoin de connaître ce qu'il contient exactement. Seul ce que représente ce type (une chaîne de caractères) est important pour comprendre le sens de la fonction.

Avec `std::pair` et `std::tuple`, la situation est un peu différente des structures de données. En utilisant ces types, vous pouvez passer n'importe quelles informations, même des informations qui n'ont pas de lien logique entre elles. Même si cela semble être une plus grande liberté, cela nuit en fait à la compréhension du code. Il est donc assez rare de

trouver des fonctions utilisant `std::pair` et `std::tuple` en paramètres. A la place, il est préférable de créer une structure de données, qui aura un nom explicite et facilitera la compréhension du code.

Correspondance entre les paramètres et arguments

La liste des arguments doit correspondre à la liste des paramètres, aussi bien le nombre que les types.

main.cpp

```
#include <iostream>

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

void g(int i) {
    std::cout << "g() avec i=" << i << std::endl;
}

int main() {
    f();    // ok
    f(123); // erreur, trop d'argument

    g();    // erreur, pas assez d'argument
    g(123); // ok
}
```

Messages d'erreur des compilateurs

Lorsqu'un compilateur trouve une fonction qui pourrait correspondre à un appel de fonction (avec le même nom de fonction), mais dont le nombre d'arguments ou les types ne correspondent pas, il produit généralement un message d'erreur indiquant qu'il ne trouve pas de fonction correspondante (“no matching function”) et donne la liste des fonctions qui pourraient convenir.

Il est donc très important de bien lire les messages du compilateur, cela facilite grandement la résolution des problèmes, en indiquant où chercher.

Par exemple, un message d'erreur classique produit par le compilateur Clang dans cette situation :

```
main.cpp:13:5: error: no matching function for call to 'f'
    f(123);
    ^
main.cpp:3:6: note: candidate function not viable:
requires 0 arguments, but 1 was provided
void f() {
     ^
```

Le compilateur indique qu'il ne trouve pas de fonction qui s'appelle `f` et dont les paramètres sont compatible avec l'appel `f(123)` (“no matching function”). Il indique à la ligne suivante qu'il connait une fonction `f` qui pourrait être candidate (“candidate function”), mais qui prend zéro argument (“requires 0 arguments”), alors que l'appel utilise un argument (“but 1 was provided”).

Pour l'appel de la fonction `g`, le message est le suivant :

```
main.cpp:15:5: error: no matching function for call to 'g'
    g();
    ^
main.cpp:7:6: note: candidate function not viable: requires
single
argument 'i', but no arguments were provided
void g(int i) {
     ^
```

De la même manière, le compilateur indique qu'il ne trouve pas de fonction correspondant à l'appel `g()`, mais qu'il trouve une fonction qui se nomme `g` et qui prend un argument.

Correspondance entre les types

En plus d'avoir le nombre d'arguments dans un appel de fonction qui doit correspondre au nombre de paramètres déclarés dans une fonction, il faut aussi que les types soient exactement les mêmes ou être implicitement convertible.

Un type est “implicitement convertible” (sous entendu “par le compilateur”) lorsque le compilateur connaît une conversion entre les types et qu'il a le droit de réaliser automatiquement cette conversion.

Par exemple, le compilateur saura convertir sans problème un argument de type entier `int` en paramètre de type réel `double` ou une littérale chaîne (de type `const char*`) en paramètre de type `std::string`. Par contre, il n'est pas possible de convertir un `std::string` en `std::vector<double>` (conversion impossible) ou en `const char*` (cette conversion est possible en appelant la fonction membre `std::string::c_str`, mais le compilateur ne peut pas l'appeler implicitement : il faut que la conversion soit écrite *explicitement* par le développeur).

```
void f(double d) {
}

void g(std::string s) {
}

int main() {
    f(123);     // conversion de int entre double
    g("hello"); // conversion de const char* en string
}
```

En revanche, la conversion (par exemple) d'un argument réel en paramètre entier produira une erreur d'arrondi (comme vous avez vu lors de l'initialisation d'une variable, avec le *narrowing*).

main.cpp

```
#include <iostream>
```

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

int main() {
    f(12.34);
    const double x = 12.34;
    f(x);
}
```

affiche :

```
main.cpp:8:7: warning: implicit conversion from 'double' to
'int' changes value from 12.34 to 12 [-Wliteral-conversion]
    f(12.34);
    ~ ^~~~
main.cpp:10:7: warning: implicit conversion turns
floating-point
number into integer: 'const double' to 'int'
[-Wfloat-conversion]
    f(x);
    ~ ^
2 warnings generated.
12
56
```

Notez que le compilateur fait la différence entre la conversion d'une littérale (*-Wliteral-conversion*) et la conversion d'une valeur réelle (*-Wfloat-conversion*). Le résultat de ces arrondis est visible dans le résultat affiché, puisque seules les parties entières sont conservées.

Pour terminer, lorsque les types des arguments et des paramètres ne sont pas du tout compatible, le compilateur indique une erreur spécifique (*no known conversion*).

main.cpp

```
#include <iostream>

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

```
int main() {
    f("hello");
}
```

affiche :

```
main.cpp:8:5: error: no matching function for call to 'f'
    f("hello");
    ^
main.cpp:3:6: note: candidate function not viable: no known
conversion from 'const char [6]' to 'int' for 1st argument
void f(int i) {
     ^
1 error generated.
```

Retour de fonction

Le paramètre de retour de fonction est une valeur qui est transmission uniquement depuis la fonction vers le code appelant. Pour rappel, la signature générale d'une fonction qui retourne une valeur est la suivante :

```
PARAMETRE_RETOUR NOM(...)
```

Lorsqu'une fonction ne retourne aucune valeur, le `PARAMETRE_RETOUR` est remplacé par `void` (qui n'est pas un type). Si la fonction retourne une valeur, `PARAMETRE_RETOUR` est remplacé par le type de la valeur que la fonction retourne.

```
int                f();  // retourne un entier
std::string        g();  // retourne une chaîne
std::vector<double> h();  // retourne un tableau
```

Dans le corps de la fonction, la valeur retournée est indiquée par le mot-clé `return` suivi de la valeur à retourner. Comme pour les paramètres d'entrées de fonctions, la valeur peut être une littérale, une variable, une constante ou une expression. Et le type de la valeur doit être identique ou implicitement convertible dans le type de retour de la

fonction.

```
int f() {
    return 1;  // retourne la valeur entière 1 (littérale)
}

int g() {
    const int i { 123 };
    return 1;  // retourne la valeur entière 123 (constante)
}
```

Bien sûr, rien n'interdit d'utiliser des paramètres d'entrée pour calculer la valeur à retourner. Par exemple :

```
int f(int i) {
    return ++i;  // retourne la valeur entière i+1
(expression)
}
```

Cette dernière syntaxe montre bien l'origine des termes “paramètres d'entrée” et “paramètre de sortie” : une fonction prend des valeurs, réalise des calculs dessus, puis retourne une valeur comme résultat. C'est très proche du concept de fonction en mathématique.

Fonction pure

Une fonction similaire au dernier code, qui prend uniquement des paramètres en entrée sans les modifier, puis retourne une valeur, est appelée une fonction pure en programmation fonctionnelle. Cela aura un intérêt particulier, en particulier pour comprendre le comportement d'une fonction et écrire des tests sur cette fonction. Cela sera vu dans le chapitre sur les tests unitaires.

Mais dans de nombreux cas, une fonction ne sera pas pure. Une fonction deviendra impure si elle :

- modifie les paramètres en entrée (ce qui est possible en utilisant des indirections, qui seront vu dans le chapitre suivant) ;
- en accédant à des variables externes à la fonction (des variables

globales).

La simple utilisation de `std::cout` est suffisant pour rendre une fonction impure.

La valeur retournée par une fonction est directement accessible dans la code appelant la fonction. L'appel de la fonction peut être directement utilisé comme valeur, ce qui permet de l'affecter à une variable ou l'utiliser dans une expression. (En fait, un appel de fonction EST une expression).

main.cpp

```
#include <iostream>

int f() { return 123; }
int g(int i) { return ++i; }

int main() {
    // initialisation dans une variable
    const auto i = f();

    // utilisation dans une expression
    std::cout << (f() + g(123)) << std::endl;
}
```

Une fonction qui retourne une valeur doit obligatoirement avoir au moins un `return`. Si ce n'est pas le cas, le compilateur produit un message d'erreur.

```
main.cpp:4:1: warning: control reaches end of
non-void function [-Wreturn-type]
}
^
```

(Ce qui signifie “atteint la fin de la fonction qui n'est pas void”).

Il est possible d'avoir plusieurs `return` dans une fonction. Lorsque la fonction arrive à un `return`, elle se termine immédiatement et la suite du code appelant est exécuté.

```
int f() {
    return 123;
    const int i { 456 };
    return i;
}
```

Le code à la suite du premier `return` n'est jamais exécuté et la fonction retourne toujours 123. (Cela n'est pas très utile dans un code aussi simple d'avoir plusieurs `return`, cela sera intéressant quand vous concevrez des algorithmes plus complexes).

Le mot-clé `return` peut également être utilisé avec une fonction qui ne retourne aucune valeur. Dans ce cas, `return` n'est suivi d'aucune valeur.

```
void f(int i) {
    return;
    ++i;
}
```

La fonction main
Un cas particulier avec la fonction `main` : vous avez peut être réalisé que la signature de cette fonction indique qu'elle doit retourner une valeur entière `int`. Mais dans les codes d'exemple de cours, le mot-clé `return` n'a jamais été utilisé jusqu'à présent.

La raison est que la fonction `main` est la seule exception à cette obligation de toujours avoir au moins un `return` dans une fonction qui retourne une valeur. Dans la cas de la fonction `main`, si `return` n'est pas présent, la fonction `main` retourne par défaut la valeur 0 (qui indique que le programme s'est déroulé correctement).

Valeurs par défaut

Vous avez vu que pour appeler une fonction, il est nécessaire de fournir autant d'arguments lors de l'appel qu'il y a de paramètres dans la déclaration de la fonction. En fait, ce n'est pas tout à fait vrai. Il est possible de fournir des paramètres par défaut lors de la déclaration d'une

fonction. Lors de l'appel, ces paramètres seront optionnels et il sera possible de ne pas fournir d'arguments pour ceux-ci.

Pour donner un paramètre par défaut à une fonction, la syntaxe est assez proche d'une initialisation de variable. La différence est que seule la syntaxe avec le signe `=` est acceptée.

```
TYPE PARAMETRE = VALEUR_PAR_DEFAUT
```

Par exemple, pour écrire une fonction qui peut prendre un paramètre entier optionnel.

main.cpp

```cpp
#include <iostream>

void f(int i = 0) {
    std::cout << i << std::endl;
}

int main() {
    f(123);  // appel avec un argument
    f();     // appel avec le paramètre par défaut
}
```

affiche :

```
123
0
```

Lorsqu'une fonction à plusieurs paramètres, une nouvelle règle s'ajoute : il n'est pas possible de faire suivre un paramètre avec une valeur par défaut par un paramètre sans valeur par défaut.

```cpp
void f(int i = 0, int j = 0, int k = 0);  // ok
void g(int i    , int j = 0, int k = 0);  // ok
void h(int i = 0, int j = 0, int k    );  // erreur
```

La fonction `h` possède un paramètre `k` sans valeur par défaut, alors que les paramètres `i` et `j` ont des valeurs par défaut.

La fonction `f` a trois paramètres optionnels, alors que la fonction `g` a un paramètre obligatoire (`i`) et deux paramètres optionnels (`j` et `k`).

```
f(1, 2, 3);   // i=1, j=2, k=3
f(1, 2);      // i=1, j=2, k=0
f(1);         // i=1, j=0, k=0
f();          // i=0, j=0, k=0

g(1, 2, 3);   // i=1, j=2, k=3
g(1, 2);      // i=1, j=2, k=0
g(1);         // i=1, j=0, k=0
g();          // erreur, i n'est pas optionnel
```

Surcharge de fonction

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 tache, 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 i, int j);
double add_double(double x, double y);
```

C'est une approche possible, mais si vous pensez au nombres de types que vous avez vu jusque 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).

Fonctions génériques

Pour cet exemple aussi simple, il existe en fait une meilleure approche, c'est d'utiliser des fonctions génériques. Une fonction générique (qui est appelé aussi fonctions *template*) sont des fonctions qui ne sont pas écrite

pour un type en particulier, mais pour différents types.

Les fonctions génériques simples seront vu dans la suite de ce cours. L'utilisation avancée des fonctions template sort du cadre de ce cours, c'est la méta-programmation en C++.

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)
```

Il est possible de définir plusieurs fonctions avec le même nom, mais des signatures différentes. 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 de arguments. (Vous voyez ici, encore, l'importance des types en C++).

Pour la fonction `add` :

```
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 second version de add
```

La possibilité d'écrire plusieurs fonctions avec la même nom, mais des signatures différentes, s'appelle la surcharge de fonctions.

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 générique 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.

Résolution des noms de fonctions

- fonctions exactes
- avec conversion
- ambiguité

Le compilateur commence par rechercher s'il connait une fonction avec le nom correspondant. Par exemple pour f(1), il trouve 2 fonctions : f(int) et f(long int). Ensuite il regarde si l'un des types en paramètre correspondant au type en argument. Ici, c'est le cas, il appelle donc f(int).

Si on écrit :

```
#include <iostream>

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

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

Le compilateur trouve la fonction f, mais le paramètre ne correspond pas. Il regarde s'il peut faire une conversion. Ici, oui, on peut convertir implicitement un int en long int. il convertie donc 1 en 1L et appelle f(long int).

S'il ne trouve pas de conversion possible, il lance un message d'erreur. Par exemple, si on appelle f(“du texte”), le compilateur donne :

```
main.cpp:19:5: error: no matching function for call to 'f'
    f("une chaine");
    ^
main.cpp:3:6: note: candidate function not viable: no known
conversion
from 'const char [11]' to 'int' for 1st argument
void f(int i) {
     ^
main.cpp:7:6: note: candidate function not viable: no known
conversion
from 'const char [11]' to 'long' for 1st argument
void f(long int i) {
     ^
1 error generated.
```

Ce qui signifie qu'il ne trouve aucune fonction correspond à l'appel de f(“une chaine”), mais qu'il a 2 candidat (2 fonction qui ont le même nom) mais sans conversion possible (“no known conversion”).

Au contraire, dans certain cas, il aura plusieurs possibilités, soit parce que vous déclarez par erreur 2 fonctions avec les mêmes paramètres, soit parce que le compilateur peut faire 2 conversions pour 2 types. Dans le premier cas :

```
void f() {
   std::cout << "première fonction f" << std::endl;
}

void f() {
   std::cout << "seconde fonction f" << std::endl;
}
```

produit le message :

```
main.cpp:7:6: error: redefinition of 'f'
void f(int i) {
     ^
main.cpp:3:6: note: previous definition is here
```

```
void f(int i) {
     ^
```

Quand le compilateur arrive à la ligne 7 et rencontre la seconde fonction f (qu'il connait déjà), il prévient qu'il connait déjà (“redefinition of 'f'”) et que la première version (“previous definition is here”) se trouve à la ligne 3.

L'autre cas est si plusieurs fonctions peuvent correspondent, l'appel est ambigu. Par exemple :

```
#include <iostream>

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

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

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

affiche le message d'erreur :

```
main.cpp:12:5: error: call to 'f' is ambiguous
    f(1u); // 1 est une littérale de type int
    ^
main.cpp:3:6: note: candidate function
void f(int i) {
     ^
main.cpp:7:6: note: candidate function
void f(long int i) {
     ^
```

Il existe une conversion de unsigned int vers int et vers long int. Il n'y a pas de priorité dans les conversions, le compilateur ne sait pas quelle conversion choisir et donc quelle fonction appeler. L'appel est ambuigu (“call to 'f' is ambiguous”), il trouve deux fonctions candidate (“candidate

function”).

La méthode qui permet au compilateur de trouver la fonction correspondant à une appel s'appelle la résolution des noms (name lookup)

Note sur bool

Comme cela a déjà été expliqué, certains types, dont les littérales chaînes (et plus généralement les pointeurs), sont convertissable automatiquement en booléen. Si on écrit la surcharge suivante :

```
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 on ajoute une fonction `f(const char*)`, elle sera appelée en premier. La raison est que la littérale chaîne est de type `const char*`, les fonctions seront appelée dans l'ordre suivant :

- f(const char*) : par de conversion entre l'argument et le paramètre ;
- f(bool) : conversion automatique ;
- f(string) : conversion passant par une classe.

Donc attention lorsque vous écrivez une fonction qui prend bool, elle peut prendre aussi n'importe quel pointeur.

Solution C++14 : écrire “abc”s pour créer une littérale de type string.

Détailler le name lookup

Chapitre précédent	Sommaire principal	Chapitre suivant