[Aller plus loin] La surcharge de fonctions et resolution 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 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 lhs, int rhs);
double add_double(double lhs, double rhs);
```

Note : Pour les operateurs 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 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).

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 a 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` precedente :

```
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 problematique est d'utiliser une fonction generique. Une fonction generique est simplement une fonction qui n'est pas ecrite pour un type de parametre en particulier, mais utilise l'inference de type pour determiner les types des parametres.

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

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

Vous verrez dans le prochain chapitre comment creer des fonctions generiques, ne vous pre-occupez pas trop de la syntaxe pour le moment. Sachez simplement que le type `T` dans les parametres de la fonction sera remplace 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 ecrire une fonction pour chaque type que la fonction pourra accepter. Et pour les fonctions generique, il est necessaire d'ecrire qu'une seule fois la fonction.

(La surcharge de fonctions et les fonctions generiques sont des concepts independants, ils peuvent etre utiliser ensemble pour creer des surcharges de fonctions generiques).

D'autres exemple de surcharge de fonctions

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

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

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

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

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

Ou encore, en faisant apparaitre l'expression intermediaire :

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

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

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

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

L'etude des operateurs et leur surcharge est suffisamment important pour etre detaille dans un chapitre dedie, dans la partie sur la programmation orientee objet.

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 resolution des noms est donc le processus qui permet au compilateur de determiner ce que signifie chacun de ces identifiants.

Analyse sequentielle

Vous avez vu au debut de ce cours qu'en programmation imperative, les instructions etaient executees les unes apres les autres. (dans hello_world). Le compilateur fonctionne de la meme facon : il lit le code ligne par ligne, de facon purement sequentielle. Lorsqu'il lit une ligne, il utilise les informations qu'il a appris en lisant les lignes precedentes.

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 connait deja f
    g();  // erreur, le compilateur ne connait pas encore g
}

void g() {}
```

Une exception a cette regle : le compilateur connait un certain nombre

d'identifiants par defaut, definis dans la norme C++. Ces identifiants sont reserves, il vous est interdit de les utiliser. Ce sont les mots-cles du langage. Vous connaissez deja :

- `auto` : inference de type ;
- `bool` : type booleen ;
- `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-cle pour les variables, mais il sera utilise aussi pour les fonctions membres) ;
- `constexpr` : expression constante (evaluee lors de la compilation si possible) ;
- `decltype` : inference de type ;
- `double` : type de nombre a virgule flottante, generalement sur 64 bits ;
- `enum` : enumeration ;
- `false` : valeur booleenne “faux” ;
- `float` : type de nombre a virgule flottante, generalement sur 32 bits ;
- `int` : type d'entier, intermediaire entre `short int` et `long int`, generalement sur 32 bits ;
- `long` : modificateur de type, permet d'utiliser des entiers (`int`) et reels (`double`) de plus grande taille ;
- `operator` : pour definir un operateur (fonction particulier, 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 signe ;
- `sizeof` : operateur permettant de connaitre la taille en memoire d'un type ou d'une variable ;
- `struct` : pour definir une structure de donnees ;
- `template` : pour definir une fonction ou une classe generique ;
- `true` : valeur booleenne “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-cles ont des utilisations que vous n'avez pas encore vues et il existe d'autres mots-cles que vous verrez par la suite. La liste complete des mots-cles est indiquee dans la documentation : C++ keywords.

Directive de pre-processeur

Dans le code du programme “hello world”, la premiere instruction rencontree par le compilateur est la directive de pré-processeur `#include`. Il existe plusieurs directives de compilation, elles seront expliquees dans un chapitre dedie.

Une directive commence toujours pas un diese `#`. La directive `#include` permet d'utiliser les fonctionnalites declarees 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-tete).

Il est possible de mettre la directive `#include` a n'importe quel endroit (en dehors des fonctions), mais la regle sequentielle s'applique : vous ne pouvez pas utiliser une fonctionnalite d'un fichier d'en-tete avant d'avoir include 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 eviter les erreurs, il est classique de placer les directives `#include` au debut du code. Cette regle sera utilisee 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 egalement comprendre que l'identifiant `main` est la definition d'une nouvelle fonction et qu'il ne connait pas encore ce nom, il va donc l'ajouter a sa liste des noms connus.

Le compilateur arrive ensuite a l'identifiant `std`. Il sait que les doubles deux-points `::` correspondent a l'operateur de portee, qui est utilise avec les espaces de noms, les classes ou les enumerations par exemple. L'espace de noms `std` est defini dans le fichier d'en-tete `iostream`, donc le compilateur connait deja cet identifiant quand il arrive a l'instruction `std::cout`.

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

La portee permet d'utiliser un meme identifiant dans plusieurs definitions, a partir du moment ou ces identifiants sont dans des portees differentes. Par exemple, la fonction `size` peut correspondre a des fonctions differentes, dans ces classes differentes : `std::string::size`, `std::vector::size`, etc. (Il est possible d'avoir plusieurs portees les unes dans les autres).

Dans le chapitre 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 connait 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 definitions pouvant correspondre a `cout` : la fonction `cout` definie dans le code et le flux de sortie `std::cout` definie dans `iostream`. Ces deux identifiants sont valides et le compilateur ne peut determiner lequel il doit utiliser, il y a ambiguite (*reference to 'cout' is ambiguous*).

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

L'utilisation de `using namespace` reduit l'interet des espaces de noms et peut poser des problemes de conflits dans les noms. L'utilisation de cette syntaxe est generalement limitee aux fichiers sources (vous verrez bientot la separation du code entre fichiers d'en-tete et fichiers source).

Regle de la definition unique et portee

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

Pour utiliser un identifiant, il faut que :

- celui-ci soit definie et pas simplement declare ;
- la definition doit etre unique (ODR, *One Definition Rule*, regle de la definition unique).

Par exemple, si vous definissez deux fois la meme fonction (meme nom et meme 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 regle de la definition unique prend en compte la portee (les blocs de code, les espaces de noms, les classes, etc.). Vous pouvez donc utiliser plusieurs fois le meme identifiant, si c'est dans des portees differentes.

```
#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.
}
```

Resolution de la surcharge

Lorsque le compilateur rencontre une fonction, il regarde dans la liste des identifiants qu'il connait pour trouver les fonctions definies avec le meme nom. Comme indique au debut de ce chapitre, il est possible d'avoir plusieurs fonctions qui possedent le meme nom, a partir du moment ou leur signature est differentes (la liste des types de leurs parametres).

Le compilateur doit donc determiner exactement quelle fonction appeler. Pour cela, il va regarder la liste des types des arguments utilises lors de l'appel de la fonction, puis selectionner la fonction la plus adaptee.

Par exemple, si vous ecrivez deux fonctions, l'une qui prend un parametre de type `int` et l'autre qui prend un parametre de type `double`. Si vous appellez cette fonction en passant une valeur de type entiere, la premiere version sera automatiquement appellee. Avec une valeur de type reelle, la seconde version sera appelllee.

main.cpp

```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 parametres correspondent parfaitement. Il n'y a aucune ambiguite sur les appels de fonction et le compilateur gere la resolution de la surcharge sans probleme.

Mais la resolution de la surcharge peut egalement fonctionner lorsque les types ne correspondent pas parfaitement. Par exemple, si vous utiliser 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.

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

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

Dans ce cas, le compilateur va generer 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) {
      ^
```

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 candidats (2 fonctions 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() {
     ^
main.cpp:3:6: note: previous definition is here
void f() {
     ^
```

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 candidates (“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

adl, cas particulier namespace, class, template...

Chapitre précédent	Sommaire principal	Chapitre suivant