La surcharge de fonctions

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 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

Plusieurs fonctions avec le même nom pose le problème de déterminer quelle fonction sera effectivement appelé 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).

Mais en fait, cette problématique de la résolution des noms est plus large (et complexe) que le simple appel de fonctions. Cela concerne plus généralement tous les identifiants.

```
int i(123);             // syntaxe alternative pour
initialiser une variable
std::string("hello");  // création d'un objet de type
std::string
void f();               // déclaration d'une fonction
g();                    // appel d'une fonction
...                     // et pleins d'autres syntaxes
```

Il existe beaucoup d'autres syntaxe utilisant des parenthèses (comme par exemple les structures de contrôles que vous verrez dans la prochaine partie) et bien sûr d'autres syntaxes sans parenthèses utilisant un identifiant.

Et bien comprendre l'ampleur du problème, il faut également rappeler qu'en plus des noms de fonctions, les noms de types sont des identifiants (souvenez vous des alias de type et des structures de données), ainsi que les noms de variables, les espaces de noms (que vous verrez par la suite), etc. Bref, pour résumer, tout est identifiant.

Déclaration et définition

- déclaration, définition, ODR - mots réservés http://en.cppreference.com/w/cpp/keyword - analyse syntaxique

ADL ?

- 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