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 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).

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).

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.

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 identifiant.

Déclaration et définition

- déclaration, définition, ODR

Analyse sequentielle

Vous avez vu au debut de ce cours qu'en programmation imperative, les instructions etaient executees les unes apres les autres. (dans Le programme "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, representant un caractere)
- class
- const
- constexpr?
- decltype
- `double`
- enum
- false
- float
- int
- long
- operator
- return
- short
- signed
- sizeof
- struct
- template
- true
- typedef
- unsigned
- using
- void

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 compilation

#include

Nom qualifié et non-qualifié

- i
- std::cout

namespace, classe, enumeration

Portee

bloc de code

parent:

```
{
    int i {};
    {
        ++i;
    }
}
```

enfant:

```
{
    {
        int i {};
    }
    ++i;
}
```

different:

```
{
    int i {};
}

{
```

```
    ++i;
}
```

fonction: portee commence des les parametres:

```
void f(int i, int j = i) {}  // ok, j utilise la valeur de i
comme
                             // paramtere par defaut, qui
est dans la meme portee
```

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 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

Chapitre précédent	Sommaire principal	Chapitre suivant