Créer ses fonctions libres

Syntaxe de base

Déjà utilisé des fonctions libres. Le prototype, c'est les algo de la STL. Le but est de mettre un bloc d'instruction que l'on va pouvoir appeler plusieurs fois

Par exemple, si on veut écrire un message deux fois :

```
std::cout << "hello, world!" << std::endl;
std::cout << "hello, world!" << std::endl;
```

On peut mettre cette ligne de code dans une fonction et appeler la fonction 2 fois.

On a déjà vu qu'une fonction était définie par :

- des informations en entrée et sortie
- un nom
- une bloc d'instructions

La syntaxe de base est similaire à la fonction main. Dans la version la plus simple, pas d'infos en entrée et sortie :

```
void nom_fonction() {
    instructions
}
```

Pour appeler cette fonction, on fait comme déjà vu :

```
nom_fonction();
```

Portée des variables

pas ici, mettre dans un chapitre avec les bloc d'instruciton

Variable sont locale à un bloc. Dès que l'on sort du bloc, la variable est détruite. Mais variable accessibles dans bloc inclus :

```
int main() {
    int i {};
    {
        std::cout << i << std::endl; // ok
    }
    std::cout << i << std::endl; // ok
}
```

Au contraire, une variable définie dans un bloc n'est pas accessible en dehors :

```
int main() {
    {
        int i {};
        std::cout << i << std::endl; // ok
    }
    std::cout << i << std::endl; // erreur, i n'existe plus
}
```

variable globale, en dehors des fonctions : fu..

Idem entre bloc :

```
int main() {
    {
        int i {};
    }
    {
        std::cout << i << std::endl; // erreur, i n'existe
pas
    }
}
```

Dans ce code, même si les 2 blocs sont des sous-bloc du bloc de la

fonction main, ce sont deux bloc différents

Plus généralement :

```
{
    int i {};
    ...  // i existe ici
}        // i est détruit ici
```

Arguments de fonctions

Même situation avec fonction :

```
void f() {
    std::cout << i << std::endl; // erreur, i n'existe pas
dans ce bloc
}

int main() {
    int i {};
    f(); // i existe lors que l'on appelle f()
}
```

Dans ce code, même si la variable i existe lorsque l'on appelle la fonction f, le bloc d'instructions de f n'est pas sous-bloc de main, donc pas accessible.

Pour envoyer des informations dans la fonction, on a déjà vu avec les lambda : utilisation de paramètres de fonction. Déclarés entre les parenthèses :

```
void f(paramètres de fonction) {
}

int main() {
    f(arguments de fonction);
}
```

Paramètres = liste de 0, 1 ou plusieurs (type + nom). Par exemple, pour

envoyer 1 valeur entière :

```
void f(int i) { // création de i
}               // destruction de i
```

Les paramètres ont même portée que variables locales, depuis le début du bloc jusqu'à la fin

Pour appeler la fonction, donne une liste d'argument, qui peut être une littérale, une variable ou une fonction qui retourne un valeur de même type. Liste des arguments doit correspondre à la liste des paramètres :

```
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
}
```

Exemple de message d'erreur pour f (clang)

```
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 f correspondant à l'appel f(123) (“no matching function”). Il indique à la ligne suivante qu'il connait une fonction f (“candidate function”), mais qui prend 0

arguments (“requires 0 arguments”) alors que l'appel donne 1 arguemnt (“but 1 was provided”)

Pour l'appel de g, le message :

```
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 g(), mais qu'il trouve une fonction g qui prend 1 argument.

On peut utiliser n'importe quel type copiable :

```
void f(int i, double d, string s, vector<int> v) {
}
```

Au besoin, le compilateur réalise une conversion pour adapter les arguments. Par exemple si une écrit une fonction qui prend un long int et qu'on passe un int :

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

int main() {
    f(123); // 123 est une littérale de type int
            // peut être converti implicitement en long int
}
```

Surcharge de fonction

(ou *overloading* ou polymorphisme ad-hoc)

Polymorphisme : plusieurs fonctions de même nom. Des fonctions peuvent avoir le même nom, tant que les paramètres sont différents :

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

void f(string s) {
    std::cout << "f(string) avec s=" << s << std::endl;
}
```

Le compilateur choisit la fonction correspondante, selon le type que l'on donne en argument :

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

    int i { 1 };
    f(i);

    long int l { 1 };
    f(l);
}
```

affiche :

```
f(int) avec i=1
f(long int) avec i=1
f(int) avec i=1
f(long int) avec i=1
```

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 arguement. 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 en 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 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 possible possible, soit parce que vous déclarez par erreur 2 fonctions avec les mêmes paramètres, soit parce que le compialteur 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 surchage 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 pas pas afficher `f(string)`, mais `f(bool)`. Si on ajoute une fonction `f(cpnst char*)` et 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

Valeur par défaut

On peut souhaiter pouvoir appeler une fonction avec et sans un argument. Par exemple f qui prend un entier ou 0 si on en donne aucune valeur

Première solution, surcharger la fonction :

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

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

int main() {
    f();    // ok, appel de f()
    f(123); // ok, appel de f(int i)
}
```

Le compilateur trouve à chaque fois deux fonctions avec le même nom, mais pas d'ambiguité pour savoir laquelle appeler.

Possibilité de simplifier en donnant une valeur par défaut à un paramètre :

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

Dans ce cas, on indique que f peut prendre un entier. Si on ne donne pas de valeur, le compilateur peut utiliser la valeur par défaut :

```
int main() {
    f();    // ok, appel de f(int i) avec i = 0
    f(123); // ok, appel de f(int i) avec i = 123
}
```

Bien sûr, il ne faut pas laisser les 2 fonctions, pour éviter les ambiguité :

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

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

int main() {
    f();    // erreur, appel de f() ou de f(int i) avec i =
0 ?
}
```

Retour de fonction

Idem dans l'autre sens :

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

int main() {
    f(); // i existe dans f()
    std::cout << i << std::endl; // erreur, i n'existe pas
dans ce bloc
}
```

Une variable déclarée localement dans une fonction ne sera pas accessible dans le code qui appelle cette fonction. Utilisation de retour de fonction, permet de retourner 1 seule valeur. Utilisation du mot-clé return

pour indiquer la valeur que la fonction doit retourner et remplacer void par le type de la valeur que l'on veut retourner.

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

int main() {
    int const j = f();
    std::cout << j << std::endl; // erreur, i n'existe pas
dans ce bloc
}
```

Lorsque l'on appelle f, la variable i dans f est créée et initialisée avec la littérale 123. après le return, la valeur de i est retournée au code appelant, la variable j est créer et initialiser en copiant la valeur retournée par la fonction f (elle copie i) tandis que la variable i est détruite.

retourner directement une valeur :

```
int f() {
    return 123;
}

int main() {
    int const j = f();
    std::cout << j << std::endl; // erreur, i n'existe pas
dans ce bloc
}
```

Portée de variable fait que l'on peut utiliser 2 variables de même noms si portée différentes. Par exemple :

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

int main() {
```

```
    int const i = f();
    std::cout << i << std::endl; // erreur, i n'existe pas
dans ce bloc
}
```

Il faut bien comprendre ici que même si les 2 variables dans la fonction f et dans main s'appelle toutes les 2 “i”, ce sont 2 variables différentes.

Mot clé return retourne immédiatement de la fonction. Si on écrit :

```
int f() {
    int const i { 123 };
    return i;
    std::cout << "on est après le return" << std::endl; //
n'est jamais exécuté
}

int main() {
    int const i = f();
    std::cout << i << std::endl; // erreur, i n'existe pas
dans ce bloc
}
```

le cout après le return n'est pas exécuté.

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++