Créer des fonctions

Programmation impérative et procédurale

Jusque maintenant, vous avez écrit du code impératif : les instructions sont exécutées une par une, dans l'ordre où elles apparaissent dans le code.

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "A" << std::endl;
    std::cout << "B" << std::endl;
    std::cout << "C" << std::endl;
    std::cout << "D" << std::endl;
}
```

affiche :

```
A
B
C
D
```

En programmation procédurale, une suite d'instructions est placée dans une procédure (on parle de “fonction” en C++) pour pouvoir être appelé plusieurs fois.

Le code suivant créé une fonction `f` qui contient 2 instructions et est appelée 2 fois.

main.cpp

```cpp
#include <iostream>
```

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

int main() {
    std::cout << "A" << std::endl;
    f();
    std::cout << "B" << std::endl;
    f();
    std::cout << "C" << std::endl;
}
```

affiche :

```
A
f_A
f_B
B
f_A
f_B
C
```

Pour suivre le déroulement d'un tel programme :

- on commence par le début de main (comme en impératif) ;
- on exécute instruction par instruction (comme en impératif) ;
- quand on arrive à un appel de fonction, on exécute les instructions de la fonction une par une ;
- quand on arrive à la fin de la fonction, on retourne au code qui a appelé la fonction et on reprend l'exécution des instructions une par une.

En réalité, les instructions dans un code C++ impératif cachent souvent des appels à des fonctions. C'est par exemple le cas des opérateurs de flux `«` et de `std::endl`. De fait, la distinction entre programmation impérative et procédurale est purement pédagogique, un code sera généralement procédurale, même si

vous n'écrivez pas explicitement des fonctions.
Ce type de programmation est encore relativement simple à suivre, il suffit de lire les instructions une par une. La difficulté vient si vous avez beaucoup de fonctions, il est pénible d'arrêter la lecture toutes les 2 lignes pour aller lire une autre partie de code. (Essayez de lire un livre et d'aller consulter un dictionnaire tous les 2 phrases, vous comprendrez).

En pratique, cela ne sera pas nécessaire si vous écrivez correctement vos fonctions, en leur donnant un nom explicite. Avec le nom de la fonction, vous pourrez comprendre ce qu'elle fait et continuer la lecture du code, sans devoir aller lire les instructions dans la fonction appellée.

Et en fait, c'est ce que vous faite depuis le début de ce cours. Vous avez déjà utiliser des fonctions, comme par exemple les algorithmes de la bibliothèque standard. Par exemple, le code suivant :

```
vector<int> v { 1, 8, 3, 5, 2, 9 };
std::sort(begin(v), end(v));
std::cout << v.front() << std::endl;
```

Lorsque vous lisez la ligne avec `std::sort`, vous comprenez que le tableau est trié et que vous afficher ensuite le premier élément. Vous n'avez pas besoin d'aller lire le code de cette fonction pour comprendre ce code et déterminé que ce code va afficher la valeur “1” (première valeur du tableau après le tri).

C'est une règle générale qui faut retenir quand vous créez vos fonctions : il faut pouvoir les comprendre et les utiliser sans devoir aller lire le code de la fonction.

Un autre exemple de fonction que vous connaissez bien maintenant : la fonction `main` :

```
int main() {
}
```

Syntaxe de base

Une fonction était définie par :

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

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

Les paramètres en entrées et sortie seront détaillés ensuite. Si pas d'information à passer, il est possible de ne pas avoir de paramètre en entrée et d'utiliser `void` comme paramètre de sortie (qui signifie “pas d'information” dans ce cas).

```
void NOM_FONCTION () {
    INSTRUCTIONS
}
```

Le nom de la fonction suit les mêmes règles que les noms des variables :

- lettres, chiffres et le caractère `_`
- majuscule ou minuscule
- ne commence pas par un chiffre
- (ne commence pas par `_' : ceci n'est pas une interdiction du langage, un nom peut commencer par _, mais c'est généralement réservé à la stl. Donc a éviter en général) Par exemple : <code> void f() {} void g() {} void h() {} void foo() {} void bar() {} void une_fonction_quelconque() {} void UneAutreFonction() {} void uNeFoNcTiOnPaSlIsIbLe() {} void f1() {} void f2() {} </code> Ces noms de fonction sont valides. Les 2 premières “f” et “g”, etc ne sont pas explicite, donc impossible de savoir ce qu'elles font sans lire leur code. Donc mauvais nom en général (mais utilisé souvent comme des codes d'explication, pour nommer des fonctions qui n'ont pas d'autre utilité que d'expliquer quelque chose ou pour lequel le nom importe peu. Ne pas utiliser dans des codes réels). Les 2 fonctions suivantes “une_fonction_quelconque” et “UneAutreFonction” ont des noms

plus explicite. Notez l'utilisation des _ et majuscule pour faciliter la lecture. Forme a préférer dans vos codes, avec des noms explicites. La suivante est un exemple de fonction valide, mais peu lisible à cause de la mauvaise utilisation des majuscules. Les 2 dernières utilise un nom avec un numéro. C'est également un exemple de nom peu explicite, à éviter dans un code réel. Exemple de noms pas valides : <code> void 123f() {} *commence par un chiffre void une&fonction() {}* caractère illégal </code> Pour appeler une fonction, simplement avec le nom. <code> nom_fonction(); </code> ===== 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 : <code> int main() { int i {}; { std::cout « i « std::endl; *ok } std::cout « i « std::endl;* ok } </code> Au contraire, une variable définie dans un bloc n'est pas accessible en dehors : <code> int main() { { int i {}; std::cout « i « std::endl; *ok } std::cout « i « std::endl;* erreur, i n'existe plus } </code> variable globale, en dehors des fonctions : fu.. Idem entre bloc : <code> int main() { { int i {}; } { std::cout « i « std::endl; *erreur, i n'existe pas } } </code> 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 : <code> { int i {}; …* i existe ici } *i est détruit ici </code> ===== Arguments de fonctions ===== Même situation avec fonction : <code> 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() } </code> 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 : <code> void f(paramètres de fonction) { } int main() { f(arguments de fonction); } </code> Paramètres = liste de 0, 1 ou plusieurs (type + nom). Par exemple, pour envoyer 1 valeur entière : <code> void f(int i) {* création de i } *destruction de i </code> 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 : <code> 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 } </code> Exemple de message d'erreur pour f (clang) <code> 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() { ^ </code> 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 : <code> 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) { ^ </code> 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 : <code> void f(int i, double d, string s, vector<int> v) { } </code> Au besoin, le compilateur réalise une conversion pour adapter les arguments. Par exemple si on écrit une fonction qui prend un long int et qu'on passe un int : <code cpp> void f(long int i) { } int main() { f(123);* 123 est une littérale de type int *peut être converti implicitement en long int } </code> ===== 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 : <code> 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; } </code> Le compilateur choisit la fonction correspondante, selon le type que l'on donne en argument : <code cpp> 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(1);* 1 est une littérale de type int f(2L); *2L est une*

littérale de type long int int i { 1 }; f(i); long int l { 2 }; f(l); } </code> affiche : <code> f(int) avec i=1 f(long int) avec i=2 f(int) avec i=1 f(long int) avec i=2 </code> 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 : <code cpp> #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 } </code> 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 : <code> 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. </code> 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 possible possible, 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 : <code> void f() { std::cout « "première fonction f" « std::endl; } void f() { std::cout « "seconde fonction f" « std::endl; } </code> produit le message : <code> 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) { ^ </code> 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 : <code cpp> #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 } </code> affiche le message d'erreur : <code> 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) { ^ </code> 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 warning>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 : <code cpp> void foo(bool) { std::cout « “f(bool)” « std::endl; } void foo(string const&) { std::cout « “f(string)” « std::endl; } foo(“abc”); </code> Ce code ne va pas afficher f(string), mais f(bool). Si on ajoute une fonction f(const 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 </note>

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'ambiguïté 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;
}
```

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 créée et initialisée 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;
}
```

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;
}
```

Il faut bien comprendre ici que même si les 2 variables dans la fonction f et dans main s'appellent 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;
}
```

le cout après le return n'est pas exécuté.

Exos

http://cpp.developpez.com/tutoriels/Explicit-C++/ecrire-algorithme-standard/

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++