

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.

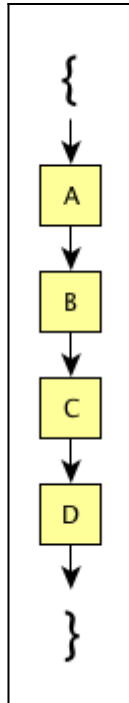
main.cpp

```
#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 crée une fonction `f` qui contient 2 instructions et est appelée 2 fois.

main.cpp

```
#include <iostream>

void f() {
    std::cout << " 1" << std::endl;
    std::cout << " 2" << std::endl;
}

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

```
std::cout << "B" << std::endl;
f();
std::cout << "C" << std::endl;
}
```

affiche :

A

1

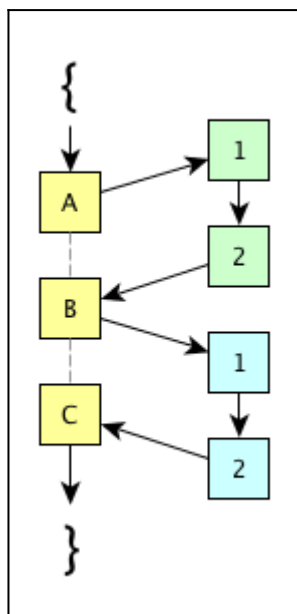
2

B

1

2

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 appelée.

Et en fait, c'est ce que vous faites depuis le début de ce cours. Vous avez déjà utilisé des fonctions, comme par exemple les algorithmes de la bibliothèque standard. Dans 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 affichez ensuite le premier élément. Vous n'avez pas besoin d'aller lire le code de cette fonction pour comprendre que ce code va afficher la valeur "1" (première valeur du tableau après le tri).

C'est une règle générale qu'il faut retenir quand vous créez une fonction :

il faut pouvoir (idéalement) la comprendre et l'utiliser sans devoir aller lire le code de la fonction ou la documentation. (Le nom de la fonction et des paramètres doit être la première documentation).

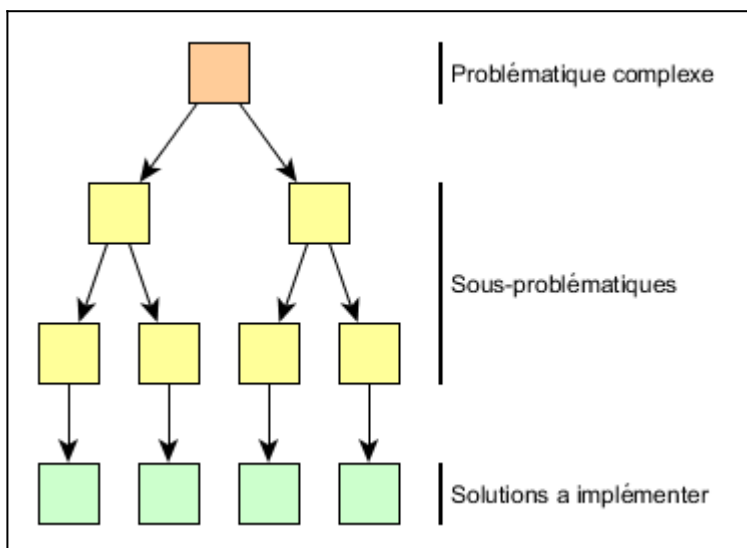
Un autre exemple de fonction que vous connaissez bien maintenant : la fonction `main` :

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

Pourquoi créer des fonctions ?

À rédiger...

Savoir utiliser des fonctions ne consiste pas seulement à connaître les syntaxes pour écrire et utiliser des fonctions. L'objectif est avant tout de savoir décomposer une problématique complexe en sous-problématiques de plus en plus simples, de façon à arriver à des solutions connues ou facilement implémentables.



Une solution sera généralement implémentée sous forme d'une fonction,

qui aura idéalement les qualités suivantes :

- courte : en quelques lignes (quelques lignes à quelques dizaines de lignes) ;
- simple à implémenter : qui utilise au maximum d'autres fonctions, en particulier des classes et algorithmes de la bibliothèque standard ;
- compréhensible : à partir du nom de la fonction et des paramètres (et de la documentation), n'importe qui peut comprendre ce que fait la fonction ;
- générique : la fonction peut être utilisée avec de nombreux types de données ;
- testable : il est facile d'écrire des tests permettant de vérifier le comportement d'une fonction.

exemple de démarche partant d'une problématique en solutions simple

Syntaxe de base

Une fonction est définie par :

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

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

Un peu de vocabulaire

La **signature** d'une fonction est le nom d'une fonction et la liste des types des paramètres. Une **déclaration** de fonction est une fonction sans implémentation (sans le bloc d'instruction, qui est remplacé par un point-virgule final). La définition d'une fonction est la fonction complète avec l'implémentation.

```
void f(int);           // déclaration
void f() { ... }       // définition
```

Une déclaration ne sert qu'à "dire" (déclarer) au compilateur qu'un identifiant existe, alors qu'une définition explique à quoi correspond un identifiant (ce qu'il est, comment il est défini).

Une même fonction peut avoir plusieurs déclarations, mais ne peut avoir qu'une seule définition (ODR, *One Definition Rule*).

Les paramètres en entrées et sortie seront détaillés ensuite. S'il n'y a 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 à éviter en général)

Par exemple :

```
void f() {}
void g() {}
void h() {}
void foo() {}
void bar() {}

void une_fonction_quelconque() {}
void UneAutreFonction() {}

void uNeFoNcTiOnPaSlIsIbLe() {}
```

```
void f1() {}  
void f2() {}
```

Note : dans ces codes, `{}` indique un bloc d'instructions... sans instructions. Donc ces fonctions ne font rien, ce sont juste des exemples pour présenter la syntaxe.

Tous ces noms de fonction sont valides. La première liste de noms (“f”, “g”, etc. jusqu’à “bar”) ne sont pas explicite, donc impossible de savoir ce qu’elles font sans lire leur code. Ce sont donc de mauvais noms 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 à 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 invalides :

```
void 123f() {}           // commence par un chiffre  
void une&fonction() {}  // caractère illégal
```

Appel de fonction

Pour appeler une fonction qui ne prend pas d’arguments et qui ne retourne pas d’information, vous pouvez l’appeler directement en utilisant son nom et des parenthèses vides. Vous avez utilisé cette syntaxe de nombreuses fois, par exemple pour la fonction `size` de `std::string` ou la fonction `clear` de `std::vector`.

```
nom_fonction();
```

Par exemple :

```
#include <iostream>

void f() {
    std::cout << "hello, world" << std::endl;
}

int main() {
    f();
}
```

affiche :

```
hello, world
```

Le compilateur lit le code dans l'ordre où il est écrit, donc, comme pour les variables, vous ne pouvez pas utiliser une fonction avant qu'elle ne soit définie. Le code suivant est donc invalide, même si la fonction existe :

```
int main() {
    f();
}

void f() {}
```

affiche le message d'erreur suivant :

```
main.cpp:2:5: error: use of undeclared identifier 'f'
    f();
    ^
```

Ce qui signifie “utilisation d'un identifiant non déclaré” (sous entendant “non déclaré dans le code qui précède la ligne de code qui utilise cet identifiant”).

Il est possible de déclarer une fonction et de la définir plus tard, en utilisant une déclaration anticipée (*forward declaration*), vous verrez cela

dans la suite de ce cours.

Vous pouvez appeler plusieurs fois une même fonction (comme vu dans le second code) ou appeler une fonction depuis une autre fonction. (C'est d'ailleurs ce que vous faites quand vous appelez une fonction quelconque depuis la fonction `main`).

```
void g() {}

void f() { g(); } // f appelle g

int main() {
    f();
    f(); // plusieurs appels de la fonction f
}
```

Une fonction qui est définie dans une autre fonction est appelée une fonction imbriquée (*nested function*). En C++, à part le cas particulier des fonctions lambdas que vous verrez ensuite, une fonction ne peut pas être imbriquée dans une autre fonction.

```
void f() {
    void g(); // declaration d'une fonction imbriquée :
    erreur
}
```

Les fonctions doivent être déclarées dans un espace de noms - qui sera vu dans le chapitre sur la création de bibliothèques - ou dans une classe - qui sera vu dans la partie programmation orientée objet. Quand vous déclarez une fonction directement en dehors de la fonction `main`, comme vous le faites depuis le début de ce chapitre, vous les déclarez en fait dans un espace de noms global et anonyme. Cela sera également vu bientôt.

Un cas particulier de fonction qui appelle une fonction : il est possible d'écrire une fonction qui s'appelle elle-même. Le code suivant est valide (tout au moins en termes de syntaxe) :

```
void f() {
    f(); // f appelle f
}
```

```
}
```

Un code qui s'appelle lui-meme est appele un code recursif. C'est une approche interessante pour resoudre de tres nombreuses problematiques. Vous verrez dans les exercices et complements quelques exemples d'algorithmes recursifs.

Cependant, le code precedent est trop simpliste et produira un crash a l'execution du programme. La raison est tres simple : la fonction `f` va appeler la fonction `f` qui va appeler la fonction `f` qui va appeler la fonction `f`... et ainsi de suite, a l'infini. Ou plus precisement, puisqu'un ordinateur n'a pas des capacites infinies, jusqu'a ce que les ressources allouees au programme soient toutes utilisees et que le programme crash.

Variable locale a une fonction

Un petit rappel sur les notions de portee et de duree de vie. Pour le moment, vous n'avez utilise que des variables locales a une fonction. Une telle fonction est declaree dans un bloc dans une fonction (dans la fonction `main` jusque maintenant, mais c'est valide pour n'importe quelle fonction que vous pouvez creer). Cette variable est utilisable a partir du moment ou elle est definie et jusqu'a la fin du bloc.

```
{  
    int i {}; // definition d'une variable i  
    ...  
} // fin de portee de la variable i
```

Les notions de portee (*scope*) et duree de vie (*lifetime*) sont tres proches (pour le moment) :

- la portee est quand la variable est utilisable dans le code ;
- la duree de vie est quand la variable existe en memoire de l'ordinateur.

Pour des variables locales, la portee et la duree de vie d'une variable sont les memes, mais vous verrez qu'il existe une autre categorie de variables (les variables dynamiques), dont la portee et la duree de vie

peuvent etre differentes.

Variables globales

Il existe en fait une troisieme categorie : les variables globales, qui ont generalement une portee globale (accessible n'importe ou dans le programme - c'est par exemple le cas de `std::cout`) et une duree de vie permanente (creation au lancement du programme et destruction lorsque le programme se termine).

L'utilisation des variables globales est problematique en termes de conception et d'utilisation, elles doivent etre evitee au maximum. C'est a dire toujours, sauf quand vous avez de tres bonnes justifications pour ne pas respecter cette regle.

Pour terminer ce rappel, les variables sont accessible dans le meme bloc ou elles sont definies et dans les blocs enfants, mais pas dans les blocs parents ou les blocs qui n'ont pas de relation hierarchique.

```
int main() {  
    {  
        int i {};  
        {  
            std::cout << i << std::endl; // ok, bloc enfant  
        }  
        std::cout << i << std::endl;      // ok, meme bloc  
    }  
    std::cout << i << std::endl;          // erreur, bloc  
parent  
    {  
        std::cout << i << std::endl;      // erreur, bloc  
independent  
    }  
}
```

Cette notion est tres importante a comprendre lorsque vous appelez des fonctions : les variables locales dans ces fonctions sont detruites a la fin de la fonction. Si vous appelez plusieurs fois une fonction, les variables locales sont creees puis detruites a chaque appel. Il n'est pas possible de

transmettre des informations entre les différents appels via des variables locales.

Considérez par exemple le code suivant.

```
#include <iostream>

void f() {
    int i;    // volontairement non initialisee
    std::cout << i << std::endl;
    i = 123;
    std::cout << i << std::endl;
}

int main() {
    f();
    f();
}
```

Si vous déroulez les appels de fonction, vous pourriez penser que le code sera le suivant (en copiant-collant le code de la fonction `f` à la place des appels de fonction) :

```
#include <iostream>

int main() {
    int i;    // volontairement non initialisee
    std::cout << i << std::endl;
    i = 123;
    std::cout << i << std::endl;

    // int i;
    std::cout << i << std::endl;
    i = 123;
    std::cout << i << std::endl;
}
```

Et donc vous pouvez vous attendre à ce que le programme affiche :

```
xxx // une valeur aleatoire quelconque, puisque i n'est pas
initialise
123
```

```
123
123
```

Mais il faut bien comprendre que ce qui se passe en realite est que chaque appel de fonction va creer une nouvelle variable locale puis la detruire. Donc meme si le nom est le meme, c'est comme si la variable etait differente. Un code plus correct serait celui ci :

```
#include <iostream>
int main() {
    int i_1;    // volontairement non initialisee
    std::cout << i_1 << std::endl;
    i_1 = 123;
    std::cout << i_1 << std::endl;

    int i_2;    // volontairement non initialisee
    std::cout << i_2 << std::endl;
    i_2 = 123;
    std::cout << i_2 << std::endl;
}
```

Avec ce code, vous comprenez bien qu'il n'y a pas de raison que les deux appels a la fonction `f` possedent la meme valeur dans `i`. Faites bien attention a cela.

Garantie des comportements

En fait, si vous executez le code propose, vous obtiendrez en fait probablement :

```
0
123
123
123
```

La raison est que le compilateur peut optimiser au mieux le programme genere a partir de votre code et peut donc utiliser le meme emplacement en memoire entre les deux appels de fonction, ce qui donnera l'impression que la valeur de `i` est transmise.

Mais bien sur, sur un code plus complexe, le comportement pourra etre différent. La norme C++ ne garantie pas que le comportement du programme sera de toujours afficher une valeur aléatoire si la variable n'est pas initialisée ou d'afficher une valeur différente entre deux appels de fonction.

Il est parfois possible d'ecrire des codes de tests, pour comprendre le comportement d'un code. Mais cela ne permet pas de savoir si c'est un comportement specifique a votre compilateur ou si c'est un comportement garantie par la norme C++.

Il est preferable d'ecrire du code dont le comportement est garantie par la norme (et donc eviter par exemple les “undefined behavior”, meme s'ils peuvent parfois avoir le comportement attendu), pour avoir un code le plus portable possible. (Meme si vous n'etes pas a l'abris qu'un compilateur particulier ne respecte pas la norme C++ sur certains points, cela peut arriver. Mais c'est suffisamment rare, surtout pour les syntaxes de bases vues dans ce cours).

Chapitre précédent	Sommaire principal	Chapitre suivant
---------------------------	---	-------------------------