

## II. Manipulation des sous-programmes

1. Structure d'un programme C++
2. Notion de sous-programmes
3. Fonctions en C++
4. Définition de fonction
5. Déclaration de fonction
6. Portée des variables
7. Paramètres formels
8. Passage de paramètres
  - 8.1 Passage par valeur
  - 8.2 Passage par adresse
    - 8.2.a Première méthode : passage par pointeur
    - 8.2.b Deuxième méthode : passage par référence
9. Notion de pile d'exécution
10. Fonctions avec résultats
11. Exercices

## 1. Structure d'un programme C++

```
#include<iostream>

using namespace std;

int main(){
    cout << "Hello, World!" << endl;
    return 0;
}
```

## 2. Notion de sous-programmes

La façon la plus naturelle pour résoudre les gros problèmes est de les diviser en une série de sous-problèmes qui peuvent être résolus (plus ou moins) indépendamment. Puis, combiner les solutions obtenues pour arriver à une solution complète.

En programmation, cette méthodologie se reflète dans l'utilisation des sous-programmes. En fait, en programmation, on décompose un programme en un ensemble de sous-programmes et on répète cette technique jusqu'à arriver à des noyaux suffisamment élémentaires pour être codés facilement.

Un sous-programme constitue un noyau élémentaire. En d'autres termes, chaque sous-programme doit réaliser une seule et unique tâche.

En C++, tous les sous-programmes sont appelés fonctions (ce qui correspond à la fois aux *fonctions* et aux *procédures* en Pascal et d'autres langages de programmation).

## 3. Fonctions en C++

Une fonction est un groupe d'instructions qui est exécuté une fois appelé.

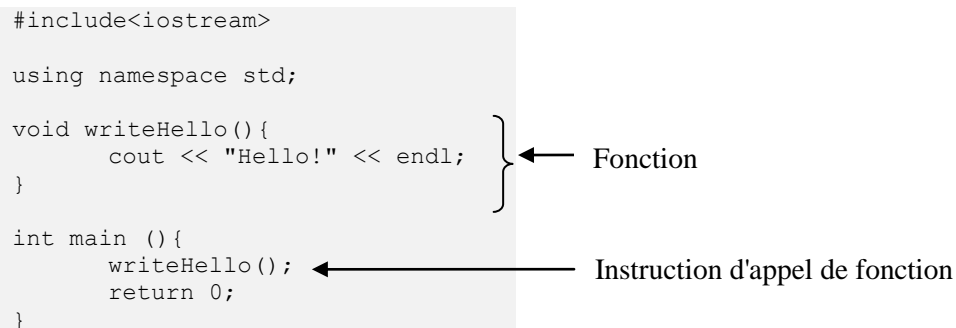
### Exemple

```
#include<iostream>

using namespace std;

void writeHello(){
    cout << "Hello!" << endl;
}

int main (){
    writeHello();
    return 0;
}
```



Définir une fonction consiste à définir un sous-programme et à lui associer un identificateur qui permet d'appeler ce sous-programme par une instruction d'appel de fonction.

L'instruction d'appel permet d'activer la fonction. Si on écrit (définit) une fonction mais on l'appelle pas, elle ne sera pas activée (exécutée). Par exemple :

```
#include<iostream>

using namespace std;

void writeHello(){
    cout << "Hello!" << endl;
}

int main (){
    cout << "Main function." << endl;
    return 0;
}
```

#### 4. Définition de fonction

La forme générale d'une définition de fonction en C++ est la suivante :

```
type identificateur(type_1 parametre_1, type_2 parametre_2, ...){
    instructions
}
```

##### Exemple 1

```
int add(int x, int y){
    ...
}
```

##### Exemple 2

```
#include <iostream>

using namespace std;

void f(){
    int x = 7;
    cout << "x = " << x << endl;
}

int main(){
    int a;
    a = 0;
    cout << "a = " << a << endl;
    f();
}
```

#### 5. Déclaration de fonction

Une déclaration de fonction indique au compilateur un nom de fonction et comment appeler cette fonction. Le corps réel de la fonction est défini séparément.

La forme générale d'une déclaration de fonction en C++ est la suivante :

```
type identificateur(type_1 parametre_1, type_2 parametre_2, ...);
```

## Exemple

```
#include <iostream>

using namespace std;

void f(); ← Déclaration de la fonction f ()

int main(){
    int a;
    a = 0;
    cout << "a = " << a << endl;
    f(); ← Appel de la fonction f ()
}

void f(){
    int x = 7;
    cout << "x = " << x << endl;
} ← Définition de la fonction f ()
```

**Remarque :** Les noms des paramètres ne sont pas obligatoires dans la déclaration d'une fonction, seuls leurs types sont requis. Par exemple, on peut écrire : `int add(int, int);`  
Au lieu de : `int add(int x, int y);`

## 6. Portée des variables

Une variable peut être locale ou globale. Une variable globale est une variable déclarée dans le corps principal du code source, en dehors de toutes les fonctions. Tandis qu'une variable locale est déclarée dans le corps d'une fonction ou d'un bloc.

```
#include <iostream>

using namespace std;

int a = 100;          // variable globale

void test(){
    int a;            // variable locale
    a = 50;
}

int main(){
    int a;            // variable locale
    a = 0;
    cout << "a = " << a << endl;
    test();
    cout << "a = " << a << endl;
}
```

### Remarques

1. Il est permis de déclarer dans une fonction (ou un bloc) un identificateur utilisé (déclaré) précédemment dans une autre fonction (ou un bloc englobant). A l'intérieur de cette fonction (ce bloc) c'est la déclaration locale qui a un sens.
2. La règle de portée des identificateurs est : *la localité masque la globalité*.
3. Ne jamais utiliser des variables globales (sauf peut être pour certaines constantes). Pourquoi? Parce que, comme nous avons déjà vu, une variable globale est accessible à l'ensemble du programme, ce

qui veut dire que n'importe quelle instruction peut modifier cette variable, donc, il devient extrêmement difficile à suivre la valeur de cette variable.

## 7. Paramètres formels

L'en-tête d'une fonction spécifie son identificateur ainsi que les identificateurs des paramètres formels s'ils existent. Par exemple :

```
int add(int a, int b);
```

Les paramètres d'une fonction procurent un mécanisme de substitution qui permet à un algorithme d'être exécuté plusieurs fois, en utilisant chaque fois des arguments différents. Les paramètres sont dits formels car les valeurs qu'ils représentent sont indéterminées jusqu'à l'appel de la fonction.

## 8. Passage de paramètres

Les paramètres formels sont définis soit par *valeur* soit par *adresse* (dit aussi par référence ou par variable).

### Exemple

```
void reduction(double x, double facteur, double& y){  
    const int CENT = 100;  
    y = x / CENT * facteur;  
}
```

Les paramètres `x` et `facteur` sont par valeur, alors que le paramètre `y` est par adresse.

**Remarque :** Tous les identificateurs définis comme paramètres formels ou déclarés à l'intérieur d'une fonction (par exemple : `x`, `facteur`, `y` et `CENT`) sont dits : locaux à celle-ci. C'est-à-dire, ils ne sont pas connus à l'extérieur de cette fonction.

### Passage par valeur

En C++, si un paramètre n'est pas précédé du symbole `&` ou `*`, ce paramètre est par valeur.

```
#include <iostream>  
  
using namespace std;  
  
void printInteger(int x){  
    cout << "Your integer = " << x << endl;  
}  
  
int main(){  
    int a = 5;  
    printInteger(a);  
}
```

← `x` : est un paramètre formel (formal parameter)

← `a` : est un paramètre actuel (actual parameter)

Un paramètre est passé par valeur lorsqu'on s'intéresse uniquement à sa valeur sans possibilité de la modifier dans le programme appelant.

## Passage par adresse

Un paramètre est dit par adresse lorsqu'on désire à la fois lire et modifier sa valeur dans le programme appelant.

### Première méthode : passage par pointeur

```
void clear(int *x){
    cout << "*x = " << *x << endl;
    *x = 0;
    cout << "*x = " << *x << endl;
}

int main(){
    int a = 5;
    cout << "a = " << a << endl;
    clear(&a);
    cout << "a = " << a << endl;
}
```

### Deuxième méthode : passage par référence

```
void clear(int& x){
    cout << "x = " << x << endl;
    x = 0;
    cout << "x = " << x << endl;
}

int main(){
    int a = 5;
    cout << "a = " << a << endl;
    clear(a);
    cout << "a = " << a << endl;
}
```

### Exemple de passage de paramètres

```
void reduction(double x, double facteur, double& y){
    const int CENT = 100;
    y = x / CENT * facteur;
}

int main(){
    double a = 10;
    double u = 0;
    reduction (a, 2, u);
    cout << "a = " << a << ", u = " << u << endl;

    reduction (5, 6, u);
    cout << "a = " << a << ", u = " << u << endl;
}
```

← Paramètres formels

← Paramètres actuels (ou arguments)

### Remarques

1. L'instruction d'appel permet d'activer la fonction en utilisant son identificateur et en lui passant une liste de paramètres actuels.

2. Un paramètre actuel passé en position *i* est substitué avec le paramètre formel occupant la position *i*.
3. Le nombre de paramètres actuels doit être égal au nombre de paramètre formels (sauf en cas d'utilisation des valeurs par défaut pour les paramètres).

```
void print(int a, int b = 20) {  
    cout << "a = " << a << ", b = " << b << endl;  
}  
  
int main () {  
    int a = 5, b = 15;  
    print(a, b);  
    print(a);  
}
```

4. Le type d'un paramètre actuel doit être compatible avec le type du paramètre formel correspondant (int, double, ...).
5. Un paramètre formel par valeur représente une variable locale dont la valeur initiale est la valeur courante du paramètre actuel. La fonction peut changer cette valeur mais ceci ne change pas la valeur du paramètre actuel.

```
void clear(int x){  
    x = 0;  
}  
  
int main(){  
    int a = 5;  
    clear(a);  
    cout << "a = " << a << endl;  
}
```

6. Si un paramètre formel est déclaré par adresse alors le paramètre actuel correspondant doit être une variable.

```
void clear(int& a){  
    a = 0;  
}  
  
int main(){  
    int a = 5;  
    clear(18); // Erreur de compilation  
    cout << "a = " << a << endl;  
}
```

7. Quand un paramètre représente un résultat, il doit être défini par adresse (il récupère le résultat).

## 9. Notion de pile d'exécution

Le rangement de données en mémoire se fait à l'aide d'une structure de pile. Une pile est une organisation logique<sup>5</sup> de la mémoire sur laquelle deux opérations seulement sont possibles :

- empiler(x) : la valeur de la variable *x* est écrite sur le sommet de la pile.

---

<sup>5</sup> qui n'existe pas réellement

- `depiler(x)` : retirer la valeur se trouvant sur le sommet de la pile et l'écrire dans la variable `x`.

### Exemple

```
void modif(int a, int* b){
    int z;
    a = 0; z = 100; *b = 0;
}

int main(){
    int z, x, y;
    x = 10, y = 20, z = 5;
    modif(x, &y);
    cout << "x = " << x << ", y = " << y << ", z = " << z << endl;
    y = 50;
    modif(x, &y);
    cout << "x = " << x << ", y = " << y << ", z = " << z << endl;
}
```

**Étape 1 - Allocation :** au début du programme (fonction `main()`), les variables `x`, `y`, `z` sont rangées sur la pile.

y	?
x	?
z	?

**Étape 2 - Initialisation :** affectation des valeurs initiales aux variables `x`, `y`, `z`.

y	20
x	10
z	5

**Étape 3 - Appel de fonction :** lors de l'activation de la fonction `modif()`, l'espace nécessaire aux paramètres formels `a` et `b` est alloué sur la pile et les valeurs de ces paramètres sont attribuées.

b		
a	10	
y	20	
x	10	
z	5	

On constate que l'espace mémoire représentant `a` contient une copie de la valeur de `x` (paramètre par valeur). Alors que l'espace de `b` contient l'adresse de la variable `y` (paramètre passé par adresse).

**Étape 4 - Exécution de la fonction :**

z	100	
b		
a	0	
y	0	
x	10	
z	5	

Durant l'exécution de la fonction `modif()`, l'espace nécessaire à la variable locale `z` est alloué sur la pile.



## Étape 5 - Fin de la fonction :

y	0
x	10
z	5

L'emplacement réservé aux variables locales (*z*) et aux paramètres formels (*a* et *b*) a été supprimé.

### Remarques

1. Lors de la deuxième activation de la fonction `modif()`, l'espace libéré par la première activation est réutilisé.
2. L'exécution de la fonction `modif()` a modifié le contenu de la variable *y* mais pas celui de la variable *x*.
3. Les valeurs des variables locales à la fonction ne sont pas conservées d'une exécution à l'autre.

## 10. Fonctions avec résultats

L'entête d'une fonction en C++ spécifie son identificateur, les paramètres formels et le type de résultat que cette fonction doit retourner.

```
type identificateur(type_1 parametre_1, type_2 parametre_2, ...){
    instructions
}
```

Lorsqu'on définit (ou on déclare) une fonction, on doit spécifier le type de résultat que cette fonction renverra. En fait, une fonction peut effectuer la tâche souhaitée sans renvoyer (retourner) un résultat (une valeur). Dans ce cas, le type de retour `type` doit être le mot-clé `void` (voir les exemples ci-dessus). Dans le cas où une fonction doit retourner un résultat, cette opération doit être faite avec le mot-clé `return` dans le corps de la fonction.

### Exemple

```
int somme(int n){
    return (n * (n + 1) / 2);
}

int main(){
    int r = somme(5);
    cout << "r = " << r << endl;
}
```

### Remarques

1. La définition d'une fonction qui retourne un résultat définit un sous-programme qui calcule une valeur unique de type scalaire (`int`, `double`, ...), pointeur, etc.
2. Il faut mettre le résultat de la fonction dans une variable de même type que le type de retour de la fonction. Par exemple : `int r = somme(5);`
3. En effet, l'identificateur d'une telle fonction permet son activation en apparaissant comme constituant d'une expression. Par exemple : `int r = 100 + somme(5) * 2;`
4. Dans le cas d'une fonction qui retourne un résultat, il doit y avoir au moins une instruction `return` dans le corps de cette fonction.

## 11. Exercices

1. Écrire une fonction qui augmente son argument par 1.
2. Écrire une fonction qui permet d'échanger les valeurs de deux variables entières.
3. Écrire une fonction qui calcule la somme des nombres entiers de 1 à N.  
 $S = 1 + 2 + 3 + \dots + N$
4. Écrire une fonction appelée `largestDigit()`, qui retourne le plus grand chiffre de son argument entier.  
Par exemple:  
    `largestDigit(3185)` retourne 8.  
    `largestDigit(-65665)` retourne 6.  
La signature de la fonction est : `int largestDigit(int n);`
5. Écrire une fonction qui prend un tableau d'entiers comme argument et retourne le plus grand nombre dans ce tableau.
6. Écrire une fonction qui prend un tableau d'entiers non négatifs comme argument et retourne le deuxième plus grand nombre entier dans ce tableau. La fonction retourne -1 s'il n'y a pas de deuxième plus grand nombre dans le tableau.

Si le tableau est	La fonction retourne
{1, 2, 3, 4}	3
{4, 1, 2, 3}	3
{1, 1, 2, 2}	1
{1, 1}	-1
{1}	-1
{}	-1