Chapitre 6 : Les Fonctions

Objectifs pédagogiques :

- Comprendre l'utilité des fonctions pour structurer un programme
- Maîtriser la différence entre prototype, définition et appel
- Savoir passer des paramètres de différentes manières (valeur, référence, pointeur)
- Utiliser la surcharge pour créer des fonctions adaptées à différents types
- Appliquer ces concepts à des problèmes d'automatique (calculs de lois de commande, filtres, etc.)

1. Introduction aux fonctions

1.1. Pourquoi utiliser des fonctions ?

En automatique, nous manipulons constamment des opérations répétitives :

- Calcul d'une loi de commande PID
- Application d'un filtre numérique
- Conversion d'unités (degrés ↔ radians, volts ↔ physiques)
- Calcul de statistiques sur des mesures

Les fonctions permettent de :

- Éviter la duplication de code : écrire une fois, utiliser partout
- Structurer le programme : découper un problème complexe en sous-problèmes simples
- Faciliter la maintenance : modifier à un seul endroit
- Permettre la réutilisation : créer des bibliothèques de fonctions

2. Syntaxe, définition, prototype et appel

2.1. Syntaxe

Type Nom_de_la_fonction(paramètres)

2.2. Définition

```
#include <iostream>
using namespace std;

// DÉFINITION COMPLÈTE de la fonction
double carre(double x) {
    double resultat = x * x;
    return resultat;  // Retourne la valeur calculée
}

int main() {
    double nombre = 5.0;
```

```
    double resultat = carre(nombre);  // APPEL de la fonction

    cout << "Le carré de " << nombre << " est " << resultat << endl;

    return 0;
}
```

2.3. Prototype (déclaration)

Le prototype annonce l'existence d'une fonction sans donner son code. Il se termine par un point-virgule.

```
#include <iostream>
using namespace std;

// PROTOTYPES (déclarations)
double carre(double x);
double cube(double x);
double puissance_quatre(double x);

int main() {
    double x = 3.0;
    cout << "carré = " << carre(x) << endl;
    cout << "cube = " << cube(x) << endl;
    cout << "puissance 4 = " << puissance_quatre(x) << endl;
    return 0;
}

// DÉFINITIONS (après main)
double carre(double x) {
    return x * x;
}

double cube(double x) {
    return x * x * x;
}

double puissance_quatre(double x) {
    return carre(x) * carre(x);  // On peut appeler d'autres fonctions
}
```

Pourquoi utiliser des prototypes ?

- Permet d'organiser le code : main() en premier, définitions après
- Indispensable pour les fonctions qui s'appellent mutuellement
- Utile pour les grands projets avec plusieurs fichiers

2.4. Règles importantes

```
// ERREUR : appel avant définition et sans prototype
int main() {
    double y = f(5.0);  // ERREUR : f n'est pas déclarée
    return 0;
}

double f(double x) {
    return x * 2;
}
// ------------------------------------------------
// CORRECT : prototype avant utilisation
double f(double x);  // Prototype

int main() {
    double y = f(5.0);  // OK
    return 0;
}

double f(double x) {
    return x * 2;
}
```

3. Passage d'arguments

3.1. Passage par valeur (copie)

Principe : La fonction reçoit une copie de l'argument. L'original n'est pas modifié.

```
#include <iostream>
using namespace std;

void modifier_valeur(double x) {
    x = x * 2;  // Modifie la copie locale
    cout << "Dans fonction : x = " << x << endl;
}

int main() {
    double a = 10.0;
```

```
    cout << "Avant appel : a = " << a << endl;
    modifier_valeur(a);
    cout << "Après appel : a = " << a << endl;  // a est toujours 10.0 !

    return 0;
}
```

3.2. Passage par référence

Principe : La fonction reçoit une référence vers l'argument. Elle peut modifier l'original.

```
#include <iostream>
using namespace std;

void modifier_reference(double& x) {  // Note le &
    x = x * 2;  // Modifie directement la variable originale
    cout << "Dans fonction : x = " << x << endl;
}

int main() {
    double a = 10.0;

    cout << "Avant appel : a = " << a << endl;
    modifier_reference(a);
    cout << "Après appel : a = " << a << endl;  // a = 20.0 maintenant !

    return 0;
}
```

3.3. Passage par pointeur

Principe : La fonction reçoit l'adresse de l'argument. C'est une autre façon de modifier l'original. Moins fréquent en C++ moderne (on préfère les références).

```
#include <iostream>
using namespace std;

void modifier_pointeur(double* x) {  // Reçoit un pointeur
    *x = *x * 2;  // Déréférencement pour modifier
    cout << "Dans fonction : *x = " << *x << endl;
}

int main() {
    double a = 10.0;
```

```
    cout << "Avant appel : a = " << a << endl;
    modifier_pointeur(&a);  // Passe l'adresse de a
    cout << "Après appel : a = " << a << endl;  // a = 20.0

    return 0;
}
```

3.4. Tableau comparatif

Mode de passage	Syntaxe	Modifie l'original ?	Copie	Peut être nul ?
Valeur	`void f(double x)`	Non	Oui	Non
Référence	`void f(double& x)`	Oui	Non	Non
Pointeur	`void f(double* x)`	Oui	Non	Oui

4. Passage de tableaux aux fonctions

4.1. Tableaux statiques

```
#include <iostream>
using namespace std;

// IMPORTANT : Le tableau est toujours passé par adresse (pas de copie)
void afficher_tableau(double tab[], int taille) {
    for (int i = 0; i < taille; i++) {
        cout << tab[i] << " ";
    }
    cout << endl;
}

// On peut modifier le tableau original
void multiplier_tableau(double tab[], int taille, double facteur) {
    for (int i = 0; i < taille; i++) {
        tab[i] *= facteur;
    }
}

// Syntaxe équivalente avec pointeur
void afficher_pointeur(double* tab, int taille) {
    for (int i = 0; i < taille; i++) {
        cout << *(tab + i) << " ";  // ou tab[i]
```

```
    }
    cout << endl;
}

int main() {
    double mesures[5] = {10.0, 20.0, 30.0, 40.0, 50.0};

    cout << "Original : ";
    afficher_tableau(mesures, 5);

    multiplier_tableau(mesures, 5, 2.0);

    cout << "Après multiplication : ";
    afficher_pointeur(mesures, 5);

    return 0;
}
```

Règle : Quand on passe un tableau, on perd l'information sur sa taille. Il faut donc toujours passer la taille en paramètre supplémentaire.

4.2. Tableaux multidimensionnels

```
#include <iostream>
using namespace std;

// Pour un tableau 2D, il faut spécifier toutes les dimensions sauf la première
void afficher_matrice(double mat[][3], int lignes) {
    for (int i = 0; i < lignes; i++) {
        for (int j = 0; j < 3; j++) {
            cout << mat[i][j] << "\t";
        }
        cout << endl;
    }
}

// Version générale avec pointeur (plus complexe)
void afficher_matrice_dyn(double** mat, int lignes, int colonnes) {
    for (int i = 0; i < lignes; i++) {
        for (int j = 0; j < colonnes; j++) {
            cout << mat[i][j] << "\t";
        }
        cout << endl;
    }
}
```

```
int main() {
    double matrice[2][3] = {{1, 2, 3}, {4, 5, 6}};

    cout << "Matrice 2x3 :" << endl;
    afficher_matrice(matrice, 2);

    return 0;
}
```

5. Retour de valeurs

5.1. Retour simple

```
double moyenne(double tab[], int taille) {
    if (taille <= 0) return 0.0;  // Gestion d'erreur

    double somme = 0.0;
    for (int i = 0; i < taille; i++) {
        somme += tab[i];
    }
    return somme / taille;
}
```

5.2. Plusieurs valeurs de retour (par références)

```
#include <iostream>
#include <cmath>
using namespace std;

// Retourne min, max et moyenne via des références
void statistiques(double tab[], int taille,
                  double& min, double& max, double& moyenne) {
    if (taille <= 0) return;

    min = tab[0];
    max = tab[0];
    double somme = 0.0;

    for (int i = 0; i < taille; i++) {
        if (tab[i] < min) min = tab[i];
        if (tab[i] > max) max = tab[i];
        somme += tab[i];
    }
```

```
    moyenne = somme / taille;
}

int main() {
    double mesures[5] = {12.5, 18.3, 9.7, 21.0, 15.2};
    double v_min, v_max, v_moy;

    statistiques(mesures, 5, v_min, v_max, v_moy);

    cout << "Statistiques :" << endl;
    cout << "Min = " << v_min << endl;
    cout << "Max = " << v_max << endl;
    cout << "Moy = " << v_moy << endl;

    return 0;
}
```

6. Surcharge de fonctions (overloading)

6.1. Principe

La surcharge permet d'avoir plusieurs fonctions avec le même nom mais des paramètres différents. Le compilateur choisit la bonne version en fonction des arguments.

```
#include <iostream>
#include <cmath>
using namespace std;

// Plusieurs fonctions avec le même nom
double carre(double x) {
    return x * x;
}

int carre(int x) {
    return x * x;
}

float carre(float x) {
    return x * x;
}

int main() {
    cout << "carré de 5.0 (double) : " << carre(5.0) << endl;
```

```
    cout << "carré de 5 (int) : " << carre(5) << endl;
    cout << "carré de 5.0f (float) : " << carre(5.0f) << endl;

    return 0;
}
```

6.2. Règles de surcharge

La surcharge doit différer par :

- Le nombre de paramètres
- Le type des paramètres
- L'ordre des paramètres

```
// Différents nombres
void afficher(double x) {
    cout << "Valeur : " << x << endl;
}

void afficher(double x, int precision) {
    cout.precision(precision);
    cout << "Valeur : " << x << endl;
}

// Différents types
double maximum(double a, double b) {
    return (a > b) ? a : b;
}

int maximum(int a, int b) {
    return (a > b) ? a : b;
}

// Différents ordres
void regler(double consigne, double duree) {
    cout << "Consigne = " << consigne << " pendant " << duree << "s" << endl;
}

void regler(double duree, double consigne, bool urgent) {
    cout << "Urgent : consigne = " << consigne << " pendant " << duree << "s" << endl;
}
```

6.3. Exemple concret : conversions d'unités

```cpp
#include <iostream>
using namespace std;

// Conversion degrés -> radians (version double)
double deg2rad(double deg) {
    return deg * 3.141592653589793 / 180.0;
}

// Version avec angle en degrés, minutes, secondes
double deg2rad(int deg, int min, double sec) {
    double total_deg = deg + min/60.0 + sec/3600.0;
    return total_deg * 3.141592653589793 / 180.0;
}

// Version pour un tableau d'angles
void deg2rad(double angles_deg[], double angles_rad[], int taille) {
    for (int i = 0; i < taille; i++) {
        angles_rad[i] = angles_deg[i] * 3.141592653589793 / 180.0;
    }
}

int main() {
    // Utilisation des différentes versions
    cout << "45° en rad : " << deg2rad(45.0) << endl;
    cout << "45°30'15\" en rad : " << deg2rad(45, 30, 15.0) << endl;

    double angles[3] = {0.0, 45.0, 90.0};
    double rad[3];
    deg2rad(angles, rad, 3);

    cout << "Tableau converti : ";
    for (int i = 0; i < 3; i++) {
        cout << rad[i] << " ";
    }
    cout << endl;

    return 0;
}
```

7. Paramètres par défaut

7.1. Définition

On peut donner des valeurs par défaut aux paramètres. Si l'appelant ne fournit pas ces paramètres, la valeur par défaut est utilisée.

```cpp
#include <iostream>
#include <cmath>
using namespace std;

// Paramètres par défaut
double tension_sortie(double tension_entree, double gain = 1.0, double offset = 0.0) {
    return tension_entree * gain + offset;
}

int main() {
    cout << "Cas 1 : " << tension_sortie(5.0) << endl;           // 5.0 * 1 + 0 = 5
    cout << "Cas 2 : " << tension_sortie(5.0, 2.0) << endl;      // 5.0 * 2 + 0 = 10
    cout << "Cas 3 : " << tension_sortie(5.0, 2.0, 1.5) << endl; // 5.0 * 2 + 1.5 = 11.5

    return 0;
}
```

7.2. Règles importantes

Les paramètres par défaut doivent être à la fin de la liste.

On ne peut pas sauter un paramètre (il faut tous les fournir ou aucun jusqu'au dernier).

```cpp
// CORRECT
void f(int a, double b = 1.0, char c = 'A');

// INCORRECT
void g(int a = 0, double b);  // ERREUR : paramètre sans défaut après avec défaut

// UTILISATION
f(5);            // a=5, b=1.0, c='A'
f(5, 2.5);       // a=5, b=2.5, c='A'
f(5, 2.5, 'B');  // a=5, b=2.5, c='B'
// f(5, , 'B');  // ERREUR : on ne peut pas sauter b
```

8. Fonctions inline

8.1. Principe

Le mot-clé inline suggère au compilateur de remplacer l'appel à la fonction par le code lui-même (pour éviter le coût d'appel).

```
// Fonction inline (pour les très petites fonctions)
inline double carre(double x) {
    return x * x;
}

int main() {
    double a = 5.0;
    double b = carre(a);  // Le compilateur peut remplacer par a*a directement

    return 0;
}
```

9. Fonctions et portée des variables

9.1. Variables locales

```
double fonction(double x) {
    double y = x * 2;       // y est locale à la fonction
    int compteur = 0;       // locale aussi

    for (int i = 0; i < 10; i++) {  // i est locale au bloc for
        compteur += i;
    }
    // i n'est plus accessible ici

    return y + compteur;
    // y et compteur sont détruites ici
}
```

9.2. Variables statiques locales

```
#include <iostream>
using namespace std;

void compteur_appels() {
    static int compteur = 0;  // Initialisée une seule fois
    compteur++;
    cout << "Appel n°" << compteur << endl;
}
```

```
int main() {
    compteur_appels();  // Appel n°1
    compteur_appels();  // Appel n°2
    compteur_appels();  // Appel n°3

    return 0;
}
```

9.3. Variables globales (à éviter)

```
// GLOBAL (à éviter si possible)
double facteur_echelle = 1.0;

void fonction1() {
    facteur_echelle = 2.0;  // Modifie la variable globale
}

void fonction2() {
    cout << "Facteur = " << facteur_echelle << endl;
}

int main() {
    fonction2();  // 1.0
    fonction1();
    fonction2();  // 2.0

    return 0;
}
```

Problèmes des globales :

- Toute fonction peut les modifier → difficile à déboguer
- Pas de contrôle d'accès
- Problèmes en programmation concurrente

10.Synthèse

Concept	Syntaxe	Utilisation
Prototype	`double f(int x);`	Déclaration avant utilisation
Définition	`double f(int x) { return x*x; }`	Code de la fonction
Appel	`y = f(5);`	Utilisation
Passage par valeur	`void f(double x)`	Ne modifie pas l'original
Passage par référence	`void f(double& x)`	Peut modifier l'original
Passage par pointeur	`void f(double* x)`	Peut modifier, peut être nul
Tableau en paramètre	`void f(double t[], int n)`	Toujours par adresse
Surcharge	`double max(double,double)` `int max(int,int)`	Même nom, paramètres différents
Paramètre par défaut	`void f(int, double=1.0)`	Valeur par défaut
Variable statique	`static int compteur;`	Persiste entre appels

Règles d'or :

- Toujours prototyper les fonctions avant de les utiliser.
- Préférer le passage par référence pour modifier ou éviter les copies.
- Utiliser const quand le paramètre ne doit pas être modifié.
- Éviter les variables globales.
- Documenter vos fonctions (rôle, paramètres, retour).