République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Ibn Khaldoun – Tiaret

Faculté des Sciences Appliquées

Département de Génie Électrique

Support de cours

Programmation en C++

Réalisé par

Mr. MAASKRI Moustafa

Expertise réalisée par

- **Dr. MOSTEFAOUI Sidahmed Mokhtar MCA Univ de Tiaret**
- **Dr. DAOUD Mohamed Amine MCA Univ de Tiaret**

Année universitaire 2024/2025

Avant-Propos (Introduction)

Ce polycopié est le fruit de plusieurs années d'enseignement de la matière « Informatique et Programmation» au sein du département de Génie Électrique de l'Université Ibn Khaldoun de Tiaret.

L'objectif de ce support pédagogique est d'offrir aux futurs ingénieurs en électrotechnique, automatique et électronique les bases solides nécessaires pour maîtriser le développement logiciel. Si l'ingénieur en génie électrique est avant tout un spécialiste du matériel (*Hardware*), la frontière avec le logiciel (*Software*) est aujourd'hui inexistante : systèmes embarqués, microcontrôleurs, simulation numérique et traitement du signal exigent une maîtrise rigoureuse de la programmation.

Le langage C++ a été choisi pour sa performance, sa gestion précise de la mémoire et son paradigme Orienté Objet, indispensables pour l'industrie moderne.

Ce manuel est structuré de manière progressive :

1. Il débute par les fondamentaux (architecture machine et syntaxe de base).
2. Il aborde ensuite l'algorithmique procédurale (boucles, fonctions, pointeurs).
3. Il se termine par une introduction à la Programmation Orientée Objet (POO), concept clé des systèmes complexes.

Chaque chapitre est accompagné d'exemples concrets et d'exercices corrigés, conçus pour consolider les acquis théoriques et préparer l'étudiant aux travaux pratiques.

Table des matières

Liste des figures

Chapitre 1

Introduction et Syntaxe Élémentaire

1.1. Notions Élémentaires d'Informatique

L'informatique (contraction d'**Information** et **automatique**) repose sur l'interaction entre deux composantes indissociables : le matériel et le logiciel.

1.1.1 Le Matériel (Hardware)

Selon l'architecture de Von Neumann, une machine de traitement de l'information se compose essentiellement de trois parties :

1. **Le Processeur (CPU) :** Le « cerveau» de l'ordinateur. Il contient l'Unité Arithmétique et Logique (UAL) pour les calculs et l'Unité de Commande pour gérer les instructions.
2. **La Mémoire Vive (RAM) :** Une zone de stockage temporaire et rapide. Elle contient les données et les programmes en cours d'exécution. Son contenu est perdu lorsque l'ordinateur est éteint (mémoire volatile).
3. **Les Périphériques d'Entrée/Sortie (E/S) :**
    - *Entrée :* Clavier, souris, micro (pour envoyer des données à la machine).
    - *Sortie :* Écran, imprimante, haut-parleurs (pour recevoir les résultats).

1.1.2 Le Logiciel (Software)

Le matériel seul est inerte. Il a besoin d'instructions pour fonctionner. Ces instructions sont regroupées sous forme de programmes ou logiciels. On distingue :

- **Le Système d'Exploitation (OS) :** Gère les ressources matérielles (ex: Windows, Linux, macOS).

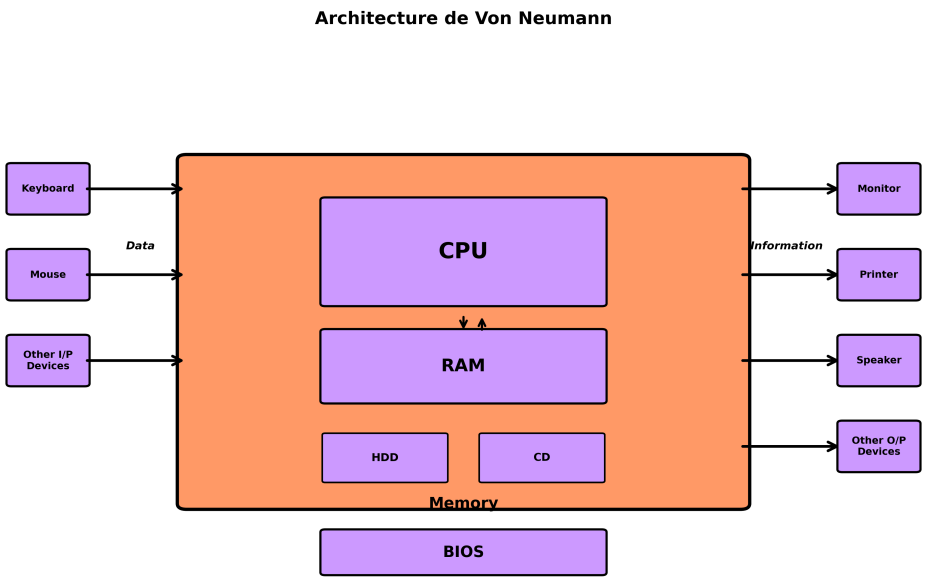

Figure 1.1: Architecture de Von Neumann

- **Les Logiciels d'Application :** Outils pour l'utilisateur (Word, Excel, Jeux, etc.).

1.2. Présentation du Langage C++

1.2.1 Le processus de compilation

Le C++ est un langage compilé. L'ordinateur ne comprend que le langage binaire (0 et 1). Pour qu'il comprenne le C++ (lisible par l'humain), on utilise un compilateur.

Le cycle de création d'un programme est le suivant :

1. **Édition :** Le programmeur écrit le Code Source (extension `.cpp`).
2. **Compilation :** Le compilateur traduit ce code en langage machine.
3. **Édition de liens (Linker) :** Il assemble le code compilé avec les bibliothèques nécessaires.
4. **Exécution :** On obtient un fichier Exécutable (extension `.exe` sous Windows).

1.2.2 Structure d'un programme C++

Voici la structure minimale d'un programme C++ :

Code extraction 1.1: Structure minimale d'un programme C++

```
// Zone d'inclusion des bibliotheques
#include <iostream>

// Utilisation de l'espace de nom standard
using namespace std;

// Fonction principale : Point d'entree du
   programme
int main() {
        // Instructions du programme
        cout << "Bienvenue en C++" << endl;

        // Fin du programme
        return 0;
}
```

- `#include <iostream>` : Indispensable pour gérer les entrées (clavier) et sorties (écran).
- `using namespace std;` : Permet d'utiliser les commandes standard (comme `cout`) sans avoir à écrire `std::cout` à chaque fois.
- `int main()` : Tout programme commence son exécution ici.
- `return 0;` : Indique au système que le programme s'est terminé sans erreur.

1.3. Syntaxe et Instructions de Base

1.3.1 Les Variables et Types

Pour stocker une information en mémoire (RAM), on doit déclarer une variable en précisant son type.

Type	Description	Exemple	Taille
`int`	Entier (sans virgule)	`int age = 20;`	4 octets
`float`	Réel (virgule flottante)	`float prix = 19.99;`	4 octets
`double`	Réel (double précision)	`double pi = 3.1415926535;`	8 octets
`char`	Caractère unique	`char lettre = 'A';`	1 octet
`bool`	Booléen (Vrai/Faux)	`bool test = true;`	1 octet

Tableau 1.1: Types de données de base en C++

Règles de nommage : Un nom de variable ne doit pas commencer par un chiffre, ne doit pas contenir d'espace ni de caractères spéciaux (sauf _), et ne doit pas être un mot réservé du langage (comme `int` ou `return`).

1.3.2 Les Entrées / Sorties (I/O)

Pour interagir avec l'utilisateur, on utilise les flux de la bibliothèque `iostream`.

- **Affichage (`cout`) :** Envoie des données vers l'écran. Le symbole « indique le sens du flux (vers la sortie).

Code extraction 1.2: Exemple d'affichage

```
cout << "Le resultat est : " << resultat << endl;
// endl permet un retour a la ligne
```

- **Saisie (`cin`) :** Récupère des données depuis le clavier. Le symbole » indique le sens du flux (vers la variable).

Code extraction 1.3: Exemple de saisie

```
int age;
cout << "Quel est votre age ? ";
cin >> age; // L'utilisateur tape une valeur qui
   est stockee dans 'age'
```

1.3.3 Les Opérateurs

- **Arithmétiques :** `+` (addition), `-` (soustraction), `*` (multiplication), `/` (division), `%` (modulo : reste de la division entière).
- **Comparaison :** `==` (égal), `!=` (différent), `<` (inférieur), `>` (supérieur), `<=` (inférieur ou égal), `>=` (supérieur ou égal).

1.3.4 Les Commentaires

Le code doit être commenté pour être compréhensible par un humain. Les commentaires sont ignorés par le compilateur.

- `//` : Commentaire sur une seule ligne.
- `/* ... */` : Commentaire sur plusieurs lignes (bloc).

Exercices Corrigés – Chapitre 1

Exercice 1.1 : Compilation Mentale

Énoncé : Que se passe-t-il si vous oubliez le ; à la fin d'une ligne ?

Correction : C'est une erreur de syntaxe. Le compilateur (étape 2) va s'arrêter et afficher une erreur (ex: « expected ';' before...»). Le fichier objet ne sera pas créé.

Exercice 1.2 : Affichage complexe

Énoncé : Écrire un programme qui affiche les guillemets à l'écran : `Il a dit "Bonjour"`.

Correction : Il faut utiliser le caractère d'échappement \ pour afficher des caractères spéciaux.

Code extraction 1.4: Solution exercice 1.2

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

int main() {
        cout << "Il a dit \"Bonjour\"" << endl;
        return 0;
}
```

Exercice 1.3 : Calculs géométriques

Énoncé : Écrivez un programme qui demande à l'utilisateur la longueur et la largeur d'un rectangle, puis affiche son périmètre et sa surface.

Correction :

Code extraction 1.5: Solution exercice 1.3

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

int main() {
        float longueur, largeur;
        float perimetre, surface;

        // 1. Saisie
        cout << "Entrez la longueur : ";
        cin >> longueur;
        cout << "Entrez la largeur : ";
        cin >> largeur;

        // 2. Traitement
        perimetre = 2 * (longueur + largeur);
```

```
        surface = longueur * largeur;

        // 3. Affichage
        cout << "Perimetre : " << perimetre << endl
           ;
        cout << "Surface : " << surface << endl;

        return 0;
}
```

Exercice 1.4 : Calculatrice Simple

Énoncé : Créez un programme qui demande deux nombres réels et affiche leur somme, produit et division.

Correction :

Code extraction 1.6: Solution exercice 1.4

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

int main() {
        double a, b;
        cout << "Entrez deux nombres : ";
        cin >> a >> b;

        cout << "Somme : " << a + b << endl;
        cout << "Produit : " << a * b << endl;

        // Attention a la division par zero
        if (b != 0)
        cout << "Division : " << a / b << endl;
        else
        cout << "Division impossible" << endl;

        return 0;
}
```

Exercice 1.5 : Échange de variables

Énoncé : Écrivez un programme qui échange le contenu de deux variables entières A et B sans utiliser de fonction, puis affichez le résultat.

Correction :

Code extraction 1.7: Solution exercice 1.5

```
int main() {
        int a = 5, b = 10, temp;
```

```
            temp = a;    // On sauvegarde a
            a = b;       // On ecrase a avec b
            b = temp;    // On restaure l'ancienne
               valeur de a dans b

            cout << a << " " << b; // Affiche 10 5
            return 0;
}
```

Chapitre 2

Structures de Contrôle

2.1. Introduction

Par défaut, un programme exécute ses instructions de manière séquentielle (ligne par ligne, du haut vers le bas). Pour créer des programmes « intelligents», nous devons pouvoir modifier cet ordre d'exécution selon certaines conditions. C'est le rôle des structures de contrôle.

2.2. Les Structures Conditionnelles

Elles permettent d'exécuter un bloc d'instructions seulement si une condition spécifique est vraie (VRAI/TRUE).

2.2.1 L'alternative simple : if / else

La structure `if` teste une condition entre parenthèses.

- Si la condition est vraie, le premier bloc est exécuté.
- Sinon (`else`), c'est le second bloc qui est exécuté (facultatif).

Syntaxe :

Code extraction 2.1: Syntaxe de if/else

```
if (condition) {
        // Instructions si la condition est VRAIE
} else {
        // Instructions si la condition est FAUSSE
}
```

Opérateurs logiques : Pour combiner plusieurs conditions, on utilise :

- `&&` (ET) : Toutes les conditions doivent être vraies.

- `||` (OU) : Au moins une condition doit être vraie.
- `!` (NON) : Inverse le résultat (Vrai devient Faux).

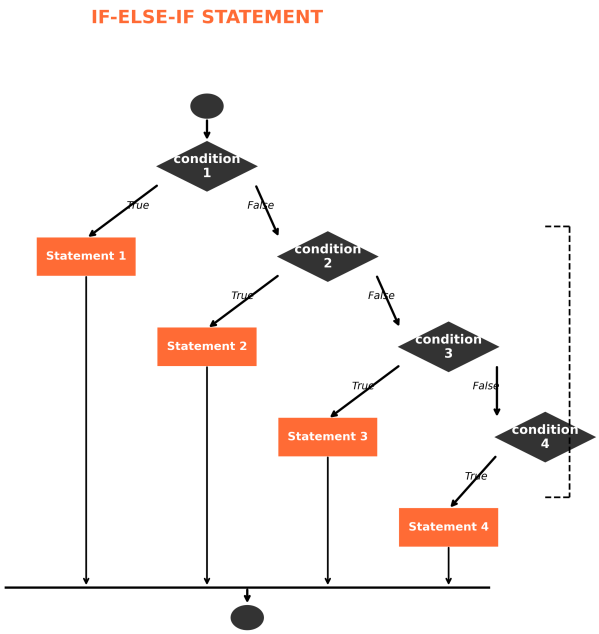

Figure 2.1: Diagramme de flux de la structure if-else-if

2.2.2 Le choix multiple : switch

Lorsque l'on doit tester une même variable contre plusieurs valeurs constantes (comme un menu de choix), le `switch` est plus lisible qu'une suite de `if... else if`.

Syntaxe :

Code extraction 2.2: Syntaxe de switch

```
switch (variable) {
        case valeur1:
        // Instructions si variable == valeur1
        break; // Sort du switch
        case valeur2:
        // Instructions si variable == valeur2
        break;
        default:
        // Instructions si aucune valeur ne
           correspond
}
```

Attention : L'instruction `break` est cruciale. Sans elle, le programme continuerait d'exécuter les instructions des `case` suivants (c'est ce qu'on appelle le « fall-through»).

2.3. Les Boucles (Structures Itératives)

Les boucles permettent de répéter un bloc d'instructions plusieurs fois. Il existe trois types de boucles en C++.

2.3.1 La boucle for (Pour)

Elle est utilisée lorsque l'on connaît à l'avance le nombre de répétitions. Elle rassemble l'initialisation, la condition et l'incrémentation sur une seule ligne.

Syntaxe :

Code extraction 2.3: Syntaxe de la boucle for

```
for (initialisation ; condition ; incrementation) {
        // Instructions a repeter
}
```

Exemple : Afficher les nombres de 0 à 9.

Code extraction 2.4: Exemple de boucle for

```
for (int i = 0; i < 10; i++) {
        cout << "i vaut : " << i << endl;
}
```

2.3.2 La boucle while (Tant que)

Elle est utilisée lorsque le nombre de répétitions n'est pas connu à l'avance. Elle répète les instructions tant que la condition est vraie. La condition est testée avant chaque passage.

Syntaxe :

Code extraction 2.5: Syntaxe de la boucle while

```
while (condition) {
        // Instructions
        // Attention : il faut modifier la
           condition ici pour eviter une boucle
           infinie
}
```

2.3.3 La boucle do...while (Faire... Tant que)

Similaire au `while`, mais la condition est testée à la fin. Cela garantit que le bloc d'instructions sera exécuté au moins une fois (utile pour la saisie de mot de passe ou les menus).

Syntaxe :

Code extraction 2.6: Syntaxe de la boucle do-while

```
do {
        // Instructions executees au moins une fois
} while (condition);
```

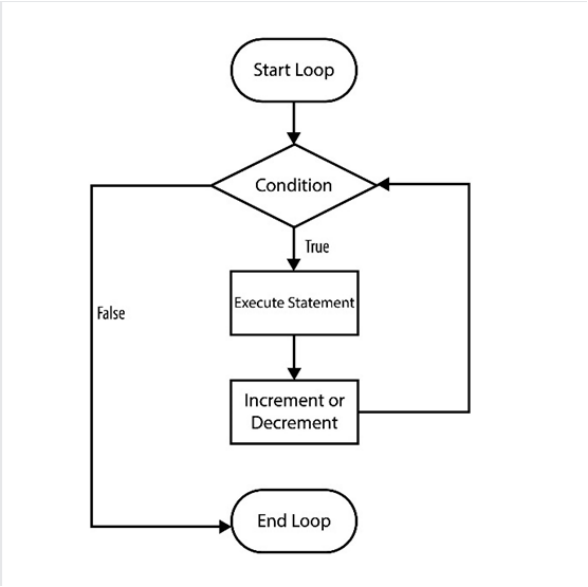

Figure 2.2: Diagramme de flux d'une boucle

2.4. Instructions de Saut (Rupture de Séquence)

Ces instructions permettent de modifier le comportement normal d'une boucle.

2.4.1 break

L'instruction `break` arrête immédiatement la boucle (ou le `switch`) et passe à l'instruction qui suit l'accolade fermante.

Exemple : Sortir d'une boucle de recherche dès que l'élément est trouvé.

2.4.2 continue

L'instruction `continue` arrête l'itération en cours et remonte directement au début de la boucle pour l'itération suivante.

Exemple : Dans une boucle de 1 à 10, on veut afficher tous les nombres sauf 5. Si `i == 5`, on fait `continue`.

Exercices Corrigés – Chapitre 2

Exercice 2.1 : Contrôle de Saisie

Énoncé : Demandez à l'utilisateur de saisir un nombre entre 1 et 10. Recommencez tant que la saisie est incorrecte.

Correction :

Code extraction 2.7: Solution exercice 2.1

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

int main() {
        int n;
        do {
                cout << "Entrez un nombre entre 1
                   et 10 : ";
                cin >> n;
        } while (n < 1 || n > 10);

        cout << "Merci, vous avez saisi : " << n <<
            endl;
        return 0;
}
```

Exercice 2.2 : Table de multiplication

Énoncé : Afficher la table de multiplication de 7.

Correction :

Code extraction 2.8: Solution exercice 2.2

```
int main() {
        int table = 7;
        for(int i = 1; i <= 10; i++) {
                cout << table << " x " << i << " =
                   " << table * i << endl;
        }
        return 0;
}
```

Exercice 2.3 : La Factorielle

Énoncé : Écrivez un programme qui demande un nombre entier positif à l'utilisateur et calcule sa factorielle (ex: $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$) en utilisant une boucle `for`. Si l'utilisateur entre un nombre négatif, affichez une erreur.

Correction :

Code extraction 2.9: Solution exercice 2.3

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

int main() {
        int n;
        long long factorielle = 1; // "long long"
           pour les grands nombres

        cout << "Entrez un entier positif : ";
        cin >> n;

        if (n < 0) {
                cout << "Erreur : Le nombre doit
                   etre positif." << endl;
        } else {
                // Boucle de calcul
                for (int i = 1; i <= n; i++) {
                        factorielle = factorielle *
                            i;
                }
                cout << "La factorielle de " << n
                   << " est " << factorielle <<
                   endl;
        }

        return 0;
}
```

Chapitre 3

Les Fonctions

3.1. Introduction et Utilité

Lorsque les programmes deviennent longs et complexes, il est inefficace d'écrire tout le code dans la seule fonction `main`.

Une fonction est un sous-programme autonome conçu pour effectuer une tâche précise (ex: calculer une moyenne, afficher un menu, vérifier un mot de passe).

Pourquoi utiliser des fonctions ?

1. **Éviter la répétition :** Si un code doit être exécuté plusieurs fois, on l'écrit une seule fois dans une fonction et on l'appelle quand on en a besoin.
2. **Modularité :** Le programme est découpé en petits blocs logiques plus faciles à comprendre et à maintenir.
3. **Débogage facilité :** Il est plus simple de tester une petite fonction isolée que de chercher une erreur dans 1000 lignes de code.

3.2. Structure d'une Fonction

Pour qu'une fonction soit utilisable, le compilateur doit connaître deux choses : son existence (le prototype) et ce qu'elle fait (sa définition).

3.2.1 Le Prototype (Déclaration)

C'est la « carte d'identité» de la fonction. Il est généralement placé avant le `main` (ou dans un fichier `.h` séparé). Il indique au compilateur le nom de la fonction, le type de donnée qu'elle renvoie, et les paramètres qu'elle accepte.

Syntaxe :

```
type_retour nom_fonction(type_param1, type_param2,
   ...);
```

3.2.2 La Définition (Le corps)

C'est le code réel de la fonction. Elle peut être placée après le `main` si le prototype a été déclaré avant.

Syntaxe complète :

Code extraction 3.1: Définition d'une fonction

```
type_retour nom_fonction(type_param1 nom_var1,
   type_param2 nom_var2) {
        // Declaration des variables locales
        // Instructions
        return valeur; // Renvoie le resultat
}
```

- **type_retour :** Le type de la variable renvoyée (`int`, `float`, etc.). Si la fonction ne renvoie rien (elle fait juste un affichage par exemple), on utilise le type spécial `void`.
- **return :** Arrête l'exécution de la fonction et renvoie la valeur au programme appelant.

3.3. Appel d'une Fonction

L'appel se fait simplement en écrivant le nom de la fonction suivi des arguments entre parenthèses.

Si la fonction renvoie une valeur, on peut stocker ce résultat dans une variable.

Exemple complet :

Code extraction 3.2: Exemple complet d'utilisation de fonction

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

// 1. Prototype
float carre(float x);

int main() {
        float nombre = 5.0;
        float resultat;

        // 2. Appel de la fonction
        resultat = carre(nombre);

        cout << "Le carre de " << nombre << " est "
            << resultat << endl;
        return 0;
}
```

```
// 3. Definition
float carre(float x) {
        float res;
        res = x * x;
        return res; // Renvoie 25.0
}
```

3.4. Portée des Variables (Locales vs Globales)

C'est un concept fondamental.

- **Variables Locales :** Une variable déclarée dans une fonction (ou dans le `main`) n'existe que dans cette fonction. Elle est détruite dès que la fonction se termine. Les autres fonctions ne peuvent pas la voir.
- **Variables Globales :** Déclarées en dehors de toute fonction (tout en haut du fichier). Elles sont accessibles partout. *Note :* Il est déconseillé d'utiliser trop de variables globales car elles rendent le code difficile à maîtriser.

3.5. Les Modes de Passage de Paramètres

Il existe deux façons de transmettre des variables à une fonction.

3.5.1 Passage par Valeur (Par défaut)

La fonction reçoit une copie de la variable.

- Si la fonction modifie cette copie, la variable originale dans le `main` ne change pas.
- **Analogie :** Vous donnez une photocopie d'un document à un collègue. S'il écrit dessus, votre original reste propre.

3.5.2 Passage par Référence (L'opérateur &)

La fonction reçoit l'adresse mémoire de la variable originale (son emplacement réel).

- Toute modification faite dans la fonction affecte directement la variable originale.
- **Analogie :** Vous donnez l'accès à votre fichier original sur le serveur. Si le collègue le modifie, vous voyez les changements.
- **Utilité :** Permet à une fonction de modifier plusieurs variables ou d'éviter de copier de grosses structures de données (optimisation).

Comparaison en code :

Passage par Valeur	Passage par Référence
`void test(int x)` `x = 10;` (La copie change) La variable du main reste inchangée.	`void test(int &x)` `x = 10;` (L'original change) La variable du main devient 10.

Tableau 3.1: Comparaison des modes de passage de paramètres

Exemple classique : La fonction echange

Pour échanger le contenu de deux variables A et B, il faut obligatoirement passer par référence, sinon l'échange ne se ferait que sur des copies temporaires.

Code extraction 3.3: Fonction d'échange avec passage par référence

```
void echange(int &a, int &b) {
        int temp = a;
        a = b;
        b = temp;
}

// Appel : echange(x, y); -> x et y sont reellement
    inverses.
```

Exercices Corrigés – Chapitre 3

Exercice 3.1 : Prédicat de parité

Énoncé : Écrire une fonction `estPair` qui prend un entier et retourne un booléen (`true` si pair, `false` sinon).

Correction :

Code extraction 3.4: Solution exercice 3.1

```
bool estPair(int n) {
        if (n % 2 == 0)
        return true;
        else
        return false;
        // Version courte : return (n % 2 == 0);
}
```

Exercice 3.2 : Division euclidienne complète

Énoncé : Écrire une fonction qui prend un dividende et un diviseur, et qui « renvoie» à la fois le quotient et le reste (via des références).

Correction :

Code extraction 3.5: Solution exercice 3.2

```
void divEuclidienne(int a, int b, int &quotient,
   int &reste) {
        if (b != 0) {
                quotient = a / b;
                reste = a % b;
        }
}

// Appel : divEuclidienne(13, 4, q, r); -> q vaudra
    3, r vaudra 1
```

Exercice 3.3 : Échange de valeurs

Énoncé : Écrivez une fonction `echange` qui prend deux entiers en paramètres et échange leurs valeurs. Affichez les valeurs avant et après l'appel dans le `main`.

Correction :

Code extraction 3.6: Solution exercice 3.3

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

// Prototype : Notez l'utilisation de '&' pour le
   passage par reference
void echange(int &a, int &b);

int main() {
        int x = 10, y = 20;

        cout << "Avant : x = " << x << ", y = " <<
           y << endl;
        echange(x, y); // Appel de la fonction
        cout << "Apres : x = " << x << ", y = " <<
           y << endl;

        return 0;
}

// Definition
void echange(int &a, int &b) {
        int temp = a;
        a = b;
        b = temp;
}
```

Chapitre 4

Tableaux et Pointeurs

4.1. Les Tableaux (Arrays)

Jusqu'ici, nous utilisions des variables simples (une variable = une valeur). Mais comment stocker les notes de 100 étudiants sans créer 100 variables `note1`, `note2`, etc. ? La réponse est : les tableaux.

4.1.1 Définition

Un tableau est une structure de données permettant de stocker plusieurs valeurs de même type dans des cases mémoire contiguës (les unes à côté des autres).

4.1.2 Les Tableaux Unidimensionnels (Vecteurs)

C'est une liste simple d'éléments.

- **Déclaration :** `type nom_tableau[taille];`
  - `int notes[5];` : Crée un tableau de 5 entiers.
- **Accès :** On accède à un élément via son indice (index) entre crochets.
  - **Attention :** En C++, les indices commencent toujours à 0.
  - Le premier élément est `notes[0]` et le dernier est `notes[4]`.
- **Initialisation :**

```
int tab[3] = {10, 20, 30}; // Remplit directement
   les cases
```

4.1.3 Les Tableaux Multidimensionnels (Matrices)

On peut créer des « tableaux de tableaux», souvent utilisés pour représenter des grilles ou des matrices.

- **Déclaration :** `type nom[lignes][colonnes];`

- `float matrice[3][4];` : Un tableau de 3 lignes et 4 colonnes.

4.2. Les Pointeurs

C'est souvent la partie redoutée des étudiants, mais c'est la plus puissante du C++.

4.2.1 Concept

Une variable classique contient une valeur (ex: `a = 5`). Un pointeur est une variable spéciale qui contient l'adresse mémoire d'une autre variable.

4.2.2 Les Opérateurs

- **L'opérateur d'adresse (&) :** Permet de connaître où une variable est stockée.
  - `&a` : Donne l'adresse de la variable `a` (ex: `0x7ffee4`).
- **L'opérateur de déréférencement (*) :** Permet d'accéder au contenu de la case pointée.
  - Si `p` contient l'adresse de `a`, alors `*p` donne la valeur de `a`.

Exemple :

Code extraction 4.1: Exemple d'utilisation des pointeurs

```
int a = 10;
int *p; // Declaration d'un pointeur sur entier

p = &a; // p pointe maintenant sur a (p contient l'
   adresse de a)
cout << *p; // Affiche 10 (la valeur pointee)

*p = 20; // Modifie la valeur de a via le pointeur
cout << a; // Affiche 20
```

4.3. Relation entre Tableaux et Pointeurs

En C++, tableaux et pointeurs sont intimement liés. Le nom d'un tableau est en réalité un pointeur constant vers son premier élément.

Si on a `int T[5];` :

- `T` est équivalent à `&T[0]` (adresse du premier élément).
- `*T` est équivalent à `T[0]` (valeur du premier élément).

Arithmétique des pointeurs : On peut se déplacer dans un tableau en ajoutant des entiers à un pointeur.

- `*(T + 1)` est équivalent à `T[1]`.
- `*(T + i)` est équivalent à `T[i]`.

4.4. Allocation Dynamique de Mémoire

Dans un tableau classique (`int T[10]`), la taille est fixée avant la compilation (allocation statique). Si on veut décider de la taille pendant l'exécution (ex: demander à l'utilisateur « Combien d'élèves y a-t-il ?»), on doit utiliser l'allocation dynamique.

4.4.1 L'opérateur new

Il demande au système de réserver de la mémoire.

```
int n;
cin >> n;
int *tab = new int[n]; // Cree un tableau de taille
    n
```

4.4.2 L'opérateur delete

En C++, la mémoire n'est pas nettoyée automatiquement. Si vous allouez de la mémoire avec `new`, vous devez la libérer avec `delete` quand vous n'en avez plus besoin, sinon vous créez une « fuite de mémoire» (*memory leak*).

```
delete[] tab; // Libere la memoire du tableau
```

Exercices Corrigés – Chapitre 4

Exercice 4.1 : Moyenne

Énoncé : Saisir 5 notes dans un tableau et calculer la moyenne.

Correction :

Code extraction 4.2: Solution exercice 4.1

```
int main() {
        float notes[5], somme = 0;

        for(int i=0; i<5; i++) {
                cin >> notes[i];
                somme += notes[i];
```

```
        }

        cout << "Moyenne : " << somme / 5.0 << endl
           ;
        return 0;
}
```

Exercice 4.2 : Recherche séquentielle

Énoncé : Écrire un programme qui demande un nombre X et vérifie s'il est présent dans un tableau T initialisé.

Correction :

Code extraction 4.3: Solution exercice 4.2

```
int main() {
        int T[5] = {12, 5, 8, 9, 1};
        int x;
        bool trouve = false;

        cin >> x;

        for(int i=0; i<5; i++) {
                if(T[i] == x) {
                        trouve = true;
                        break; // On arrete de
                           chercher
                }
        }

        if(trouve)
        cout << "Trouve !";
        else
        cout << "Absent.";

        return 0;
}
```

Exercice 4.3 : Manipulation de base

Énoncé : Déclarez un entier, un pointeur sur cet entier, et modifiez la valeur de l'entier en passant par le pointeur.

Correction :

Code extraction 4.4: Solution exercice 4.3

```
int main() {
        int x = 10;
```

```
        int *p = &x;

        cout << *p << endl; // Affiche 10
        *p = 50;
        cout << x << endl; // Affiche 50

        return 0;
}
```

Exercice 4.4 : Tableau dynamique

Énoncé : Demandez à l'utilisateur la taille d'un tableau, allouez-le, remplissez-le avec les carrés des indices (0, 1, 4, 9...), affichez-le, puis libérez la mémoire.

Correction :

Code extraction 4.5: Solution exercice 4.4

```
int main() {
        int n;
        cout << "Taille ? ";
        cin >> n;

        int *tab = new int[n]; // Allocation

        for(int i=0; i<n; i++)
        tab[i] = i * i; // Remplissage

        for(int i=0; i<n; i++)
        cout << tab[i] << " "; // Affichage

        delete[] tab; // Liberation
        return 0;
}
```

Chapitre 5

Les Fichiers

5.1. Introduction

Jusqu'à présent, toutes les données manipulées par nos programmes étaient stockées dans la RAM (mémoire vive). Le problème de la RAM est qu'elle est volatile : dès que le programme s'arrête ou que l'ordinateur s'éteint, toutes les données sont perdues.

Pour conserver des informations de manière permanente (persistance), nous devons les écrire sur le disque dur sous forme de fichiers.

5.2. La Bibliothèque <fstream>

Pour manipuler des fichiers en C++, il faut inclure la bibliothèque spécifique :

```
#include <fstream>
```

Elle définit deux types principaux de variables (flux) :

1. `ofstream` (Output File STREAM) : Pour écrire des données vers un fichier (Sortie).
2. `ifstream` (Input File STREAM) : Pour lire des données depuis un fichier (Entrée).

5.3. Écrire dans un fichier

Le processus se fait en 3 étapes : Ouvrir, Écrire, Fermer.

Code extraction 5.1: Écriture dans un fichier

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

```
int main() {
        // 1. Declaration et Ouverture
        // On cree un flux de sortie nomme '
           monFichier' vers "test.txt"
        ofstream monFichier("E:/test.txt");

        // 2. Verification
        if (monFichier.is_open()) {
                // 3. Ecriture (comme avec cout,
                   mais vers le fichier)
                monFichier << "Bonjour tout le
                   monde." << endl;
                monFichier << "Ceci est une ligne
                   de texte." << endl;
                monFichier << 42 << endl;

                // 4. Fermeture (Obligatoire pour
                   valider l'enregistrement)
                monFichier.close();
                cout << "Ecriture terminee." <<
                   endl;
        } else {
                cout << "Erreur : Impossible d'
                   ouvrir le fichier." << endl;
        }

        return 0;
}
```

5.4. Lire un fichier

Pour lire, on utilise `ifstream`. On lit généralement ligne par ligne ou mot par mot.

Code extraction 5.2: Lecture d'un fichier

```
#include <iostream>
#include <fstream>
#include <string> // Necessaire pour stocker le
   texte lu
using namespace std;

int main() {
        ifstream lecture("E:/test.txt");
        string ligne; // Variable pour stocker la
           ligne lue
```

```
        if (lecture.is_open()) {
                // Boucle de lecture : tant qu'on n
                   'est pas a la fin du fichier
                while ( getline(lecture, ligne) ) {
                        cout << ligne << endl; //
                           Affiche la ligne lue a l
                           'ecran
                }
                lecture.close();
        } else {
                cout << "Erreur d'ouverture en
                   lecture." << endl;
        }

        return 0;
}
```

`getline(flux, variable)` : Lit une ligne entière (espaces inclus) et la place dans la variable. Renvoie Faux si la fin du fichier est atteinte.

Exercices Corrigés – Chapitre 5

Exercice 5.1 : Copie de fichier

Énoncé : Écrire un programme qui lit `source.txt` et copie son contenu ligne par ligne dans `destination.txt`.

Correction :

Code extraction 5.3: Solution exercice 5.1

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

int main() {
        ifstream src("source.txt");
        ofstream dest("destination.txt");
        string ligne;

        if(src.is_open() && dest.is_open()) {
                while(getline(src, ligne)) {
                        dest << ligne << endl;
                }
                src.close();
                dest.close();
        }
```

```
        return 0;
}
```

Exercice 5.2 : Journal de bord

Énoncé : Écrivez un programme qui demande à l'utilisateur de saisir une phrase, puis ajoute cette phrase à la fin d'un fichier texte nommé « journal.txt». Ensuite, relisez tout le fichier pour afficher son contenu à l'écran.

Correction :

Code extraction 5.4: Solution exercice 5.2

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

int main() {
        string phrase;

        // --- ECRITURE (Mode 'app' pour append/
           ajouter) ---
        ofstream fichierEcriture("journal.txt", ios
           ::app);

        if (fichierEcriture.is_open()) {
                cout << "Ecrivez une phrase a
                   ajouter au journal : ";
                getline(cin, phrase); // getline
                   pour lire avec les espaces
                fichierEcriture << phrase << endl;
                fichierEcriture.close();
        } else {
                cout << "Erreur d'ouverture en
                   ecriture." << endl;
        }

        // --- LECTURE ---
        cout << "\n--- Contenu du fichier ---" <<
           endl;
        ifstream fichierLecture("journal.txt");
        string ligne;

        if (fichierLecture.is_open()) {
                while (getline(fichierLecture,
                   ligne)) {
                        cout << ligne << endl;
```

```
                }
                fichierLecture.close();
        } else {
                cout << "Erreur d'ouverture en
                   lecture." << endl;
        }

        return 0;
}
```

Chapitre 6

Programmation Orientée Objet (POO)

6.1. Changement de Paradigme

- **Programmation Structurée (Classique) :** On sépare les données (variables) des opérations (fonctions).
  - Formule : Algorithmes + Données = Programme.
- **Programmation Orientée Objet (POO) :** On regroupe les données et les opérations qui les manipulent au sein d'une même entité : l'Objet.
  - Formule : Méthodes + Données = Objet.

Avantages : Meilleure organisation, code réutilisable, modélisation plus proche du monde réel (ex: une classe « Voiture», une classe « CompteBancaire»).

6.2. Classes et Objets

- **La Classe :** C'est le plan de construction (le moule). Elle définit de quoi l'objet sera composé.
- **L'Objet :** C'est une instance concrète de la classe (le gâteau fabriqué avec le moule).

6.2.1 Structure d'une classe

Une classe contient deux types de membres :

1. **Attributs :** Les variables (données).
2. **Méthodes :** Les fonctions (comportements).

6.3. Encapsulation (Private / Public)

C'est un principe de sécurité fondamental. On cache les détails internes de l'objet pour empêcher leur modification anarchique de l'extérieur.

- **`private :`** Accessible uniquement par les méthodes de la classe elle-même. (C'est ici qu'on met les attributs).
- **`public :`** Accessible par tout le monde (le `main`). (C'est ici qu'on met les méthodes).

6.4. Exemple Complet : La Classe Point

Voici comment créer un type « Point» géométrique capable de se gérer lui-même.

Code extraction 6.1: Exemple complet : Classe Point

```
#include <iostream>
#include <cmath> // Pour sqrt et pow
using namespace std;

// Definition de la classe (Le plan)
class Point {
        // Partie cachee (Encapsulation)
        private:
        float x, y;
        char nom;

        // Partie visible (Interface)
        public:
        // Methode pour definir les valeurs
        void saisir() {
                cout << "Nom du point : "; cin >>
                   nom;
                cout << "Abscisse x : "; cin >> x;
                cout << "Ordonnee y : "; cin >> y;
        }

        // Methode pour s'afficher
        void afficher() {
                cout << "Point " << nom << "(" << x
                    << "," << y << ")" << endl;
        }
```

```
        // Methode pour calculer la distance avec
           un autre point p
        float distance(Point p) {
                return sqrt( pow(x - p.x, 2) + pow(
                   y - p.y, 2) );
        }
};

// Programme principal
int main() {
        // Creation de deux objets (Instances)
        Point A, B;

        cout << "--- Saisie du point A ---" << endl
           ;
        A.saisir(); // On demande a l'objet A de
           lancer sa methode saisir

        cout << "--- Saisie du point B ---" << endl
           ;
        B.saisir();

        cout << "--- Affichage ---" << endl;
        A.afficher();
        B.afficher();

        // Calcul de distance
        float d = A.distance(B);
        cout << "Distance AB = " << d << endl;

        return 0;
}
```

Exercices Corrigés – Chapitre 6

Exercice 6.1 : Gestion de Compte Bancaire

Énoncé : Créez une classe `Compte` avec :

- Attributs privés : `solde`, `titulaire`.
- Constructeur : initialise le titulaire et met le solde à 0.
- Méthode `deposer(montant)`.
- Méthode `retirer(montant)` : vérifie s'il y a assez d'argent.

- Méthode `afficher()`.

  Correction :

Code extraction 6.2: Solution exercice 6.1

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

class Compte {
        private:
        string titulaire;
        float solde;

        public:
        // Constructeur
        Compte(string nom) {
                titulaire = nom;
                solde = 0.0;
        }

        void deposer(float m) {
                solde += m;
        }

        bool retirer(float m) {
                if (solde >= m) {
                        solde -= m;
                        return true;
                } else {
                        cout << "Fonds insuffisants
                            !" << endl;
                        return false;
                }
        }

        void afficher() {
                cout << "Compte de " << titulaire
                   << " : " << solde << " EUR" <<
                   endl;
        }
};

int main() {
        Compte c1("Dupont");
        c1.deposer(1000);
        c1.retirer(200);
        c1.afficher(); // Affiche 800 EUR
```

```
        return 0;
}
```

Exercice 6.2 : Classe Rectangle

Énoncé : Créez une classe `Rectangle` avec :

- Attributs privés : `largeur`, `hauteur`.
- Méthode publique `setDimensions(l, h)` pour initialiser les valeurs.
- Méthode publique `surface()` qui retourne la surface.
- Méthode publique `perimetre()` qui retourne le périmètre.

Dans le `main`, créez un objet, initialisez-le et affichez ses caractéristiques.

Correction :

Code extraction 6.3: Solution exercice 6.2

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

class Rectangle {
        private:
        float largeur;
        float hauteur;

        public:
        // Methode pour definir les valeurs (Setter
           )
        void setDimensions(float l, float h) {
                largeur = l;
                hauteur = h;
        }

        // Methode pour calculer la surface
        float surface() {
                return largeur * hauteur;
        }

        // Methode pour calculer le perimetre
        float perimetre() {
                return 2 * (largeur + hauteur);
        }
};

int main() {
        Rectangle monRect; // Creation de l'objet
        monRect.setDimensions(5.0, 3.0);
```

```
            cout << "Surface : " << monRect.surface()
               << endl;
            cout << "Perimetre : " << monRect.perimetre
               () << endl;

            return 0;
        }
```

Références et Bibliographie

Ouvrages Fondateurs (En Anglais)

1. Stroustrup, Bjarne. *The C++ Programming Language*. 4th Edition. Addison-Wesley Professional, 2013.
    - Note : Écrit par le créateur du C++, c'est la référence absolue pour les détails techniques du langage.
2. Stroustrup, Bjarne. *Programming: Principles and Practice Using C++*. 2nd Edition. Addison-Wesley, 2014.
    - Note : Plus pédagogique, souvent utilisé comme manuel d'introduction dans les universités américaines.

Ouvrages Pédagogiques (En Français)

3. Delannoy, Claude. *Programmer en langage C++*. 10ème édition. Éditions Eyrolles, 2021.
    - Note : Le livre le plus populaire dans les universités francophones pour sa clarté et ses nombreux exercices.
4. Bersini, Hugues. *La programmation orientée objet*. 7ème édition. Éditions Eyrolles.
    - Note : Excellent pour comprendre les concepts théoriques de la POO (Classes, Héritage, Polymorphisme) indépendamment du langage.

Documentation et Standards en Ligne

5. ISO/IEC 14882. Standard International pour le langage de programmation C++.
6. CppReference.com (Version FR). https://fr.cppreference.com
    - Note : Une documentation technique type « wiki», tenue à jour par la communauté et très rigoureuse. Idéale pour vérifier la syntaxe d'une fonction standard.