Cours de Programmation en C

Auteur: Loïc PORTOIS

Table des matières

Introduction au Langage C

Qu'est-ce que le langage C ?

Le langage C est l'un des langages de programmation les plus anciens et les plus influents jamais créés. Développé au début des années 1970 par Dennis Ritchie aux Bell Labs, il a été conçu dans un but précis : écrire le système d'exploitation UNIX. Depuis lors, son influence n'a cessé de croître, et il a servi de fondation à de nombreux autres langages populaires, notamment C++, C#, Java, et Python.

Le C est qualifié de langage de **programmation procédurale**. Cela signifie qu'il se base sur le concept d'appels de procédures ou de fonctions. Un programme en C est essentiellement une

collection de fonctions qui s'appellent les unes les autres pour accomplir une tâche.

C'est également un **langage compilé**. Contrairement aux langages interprétés (comme Python ou JavaScript) où le code est lu et exécuté ligne par ligne par un interpréteur, un programme C doit d'abord être entièrement traduit en langage machine par un programme spécial appelé **compilateur**. Ce processus crée un fichier exécutable que l'ordinateur peut comprendre et lancer directement. Cette compilation le rend extrêmement rapide à l'exécution.

Pourquoi apprendre le C ?

Même avec l'émergence de langages plus modernes, apprendre le C en 2025 reste une compétence extrêmement précieuse pour plusieurs raisons :

- **Performance :** Le C offre un contrôle quasi-total sur la machine. Il permet de manipuler directement la mémoire, ce qui le rend idéal pour les applications où la vitesse est critique, comme les jeux vidéo, les systèmes d'exploitation, et les logiciels embarqués (le code qui fait fonctionner votre micro-ondes ou votre voiture).
- **Compréhension fondamentale :** Apprendre le C, c'est apprendre le fonctionnement interne d'un ordinateur. Des concepts comme les pointeurs, l'allocation de mémoire et la compilation vous donnent une compréhension profonde de ce qui se passe "sous le capot".
- **Fondation du monde moderne :** Une part immense de la technologie que nous utilisons repose sur du code C ou C++. Le noyau de Windows, Linux, et macOS est écrit en grande partie en C.
- **Porte d'entrée vers d'autres langages :** Maîtriser le C facilite grandement l'apprentissage de langages dérivés comme le C++ ou l'Objective-C, et même d'autres langages qui, bien que différents, partagent des concepts syntaxiques.

Votre premier programme : "Hello, World!"

La tradition, lorsqu'on apprend un nouveau langage, est de commencer par écrire un programme qui affiche simplement "Hello, World!" à l'écran. C'est un excellent moyen de s'assurer que notre environnement de développement est correctement configuré et de voir la structure de base d'un programme.

Voici le code :

```c
#include <stdio.h>

int main() {
    // Affiche le message a l'ecran
    printf("Hello, World!\n");

    return 0;
}
```

Analyse du code, ligne par ligne

Décortiquons ce petit programme. Chaque élément a son importance.

- `#include <stdio.h>` : C'est une **directive de préprocesseur**. Avant même que la compilation ne commence, un programme appelé préprocesseur lit le code. Quand il voit une ligne commençant par `#`, il la traite. Ici, `#include` lui demande d'inclure le contenu d'un autre fichier. `<stdio.h>` est le nom d'un fichier d'en-tête (header) de la **bibliothèque standard** du C. Il signifie **STanDard Input/Output** (Entrée/Sortie Standard) et il contient les déclarations de fonctions de base pour lire et écrire du texte, comme la fonction `printf`. Sans cette ligne, le compilateur ne saurait pas ce que signifie `printf`.
- `int main()` : C'est la **fonction principale**. Chaque programme C exécutable doit avoir une et une seule fonction nommée `main`. C'est le point d'entrée de votre programme. Quand vous lancez l'exécutable, le système d'exploitation appelle cette fonction. Le `int` devant `main` indique que cette fonction "retourne" un nombre entier (*integer*) au système à la fin de son exécution.
- `{ ... }` : Les accolades définissent un **bloc de code**. Tout ce qui se trouve entre l'accolade ouvrante `{` et l'accolade fermante `}` constitue le corps de la fonction `main`.
- `// Affiche le message a l'ecran` : Ceci est un **commentaire**. Les commentaires sont ignorés par le compilateur. Ils ne servent qu'à laisser des notes dans le code pour les humains qui le lisent. En C, un commentaire sur une seule ligne commence par `//`.
- `printf("Hello, World!\n");` : C'est l'instruction qui fait le travail.
  - `printf` est une fonction (provenant de `stdio.h`) qui signifie "print formatted" (imprimer de manière formatée).
  - Les parenthèses `()` contiennent les **arguments** de la fonction. Ici, nous lui donnons le texte à afficher.
  - Le texte `"Hello, World!\n"` est une **chaîne de caractères** (string). Les guillemets doubles en marquent le début et la fin.

    - Le `\n` à la fin est une **séquence d'échappement**. Il représente un caractère invisible : le retour à la ligne (newline). Sans lui, le curseur resterait sur la même ligne après l'affichage.
    - Le point-virgule `;` à la fin est crucial. En C, chaque instruction doit se terminer par un point-virgule. C'est ainsi que le compilateur sait où une commande se termine et où la suivante commence.
- `return 0;` : Cette ligne termine la fonction `main`. Elle retourne la valeur `0` au système d'exploitation. Par convention, retourner `0` signifie que le programme s'est terminé avec succès. Une autre valeur (par exemple `1`) indiquerait qu'une erreur s'est produite.

Compiler et exécuter votre programme

Maintenant que nous comprenons le code, mettons-le en pratique.

Étape 1 : Écrire le code

Ouvrez un éditeur de texte simple (comme Notepad sur Windows, TextEdit sur Mac, ou Gedit/Nano/Vim sur Linux) et copiez-y le code "Hello, World!". Enregistrez le fichier sous le nom `hello.c`. L'extension `.c` est l'extension standard pour les fichiers de code source C.

Étape 2 : Compiler

Pour compiler, vous avez besoin d'un compilateur C. Le plus courant est **GCC** (GNU Compiler Collection). Ouvrez un terminal (ou une invite de commandes) et naviguez jusqu'au dossier où vous avez enregistré votre fichier. Ensuite, tapez la commande suivante :

```
gcc hello.c -o hello
```

Détaillons cette commande :

- `gcc` : Le nom du programme compilateur.
- `hello.c` : Le fichier source en entrée.
- `-o hello` : L'option `-o` (pour *output*) permet de spécifier le nom du fichier exécutable en sortie. Si vous omettez cette partie, le compilateur créera un fichier nommé `a.out` (sur Linux/Mac) ou `a.exe` (sur Windows).

Si tout se passe bien, la commande ne dira rien et vous verrez un nouveau fichier nommé `hello` (ou `hello.exe`) apparaître dans votre dossier.

Étape 3 : Exécuter

Dans le même terminal, tapez la commande suivante pour lancer votre programme :

```
./hello
```

(Sur Windows, vous pouvez simplement taper `hello`). Le `./` est nécessaire sur les systèmes de type UNIX pour indiquer que l'exécutable se trouve dans le dossier courant.

Vous devriez voir apparaître à l'écran :

```
Hello, World!
```

Félicitations, vous avez écrit, compilé et exécuté votre premier programme en C !

Exercices

La meilleure façon d'apprendre est de pratiquer. Essayez de résoudre ces petits problèmes.

Exercice 1 : Personnalisation

Modifiez le programme `hello.c` pour qu'il n'affiche plus "Hello, World!" mais votre nom et une petite phrase de présentation. Par exemple : "Bonjour, je m'appelle Jean et j'apprends le C.". N'oubliez pas de recompiler avant d'exécuter !

Exemple de solution :

```c
#include <stdio.h>

int main() {
    // Affiche un message personnalisé
    printf("Bonjour, je m'appelle Loïc et j'apprends le C.\n");

    return 0;
}
```

Exercice 2 : L'art de la nouvelle ligne

Écrivez un nouveau programme, appelé `poeme.c`, qui affiche les quatre premiers vers de votre poème ou chanson préférée. Chaque vers doit être sur sa propre ligne.
Indice : Vous pouvez utiliser plusieurs fois la fonction `printf`, ou n'en utiliser qu'une seule avec plusieurs séquences `\n`.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    printf("Le vent se lève ! . . . il faut tenter de vivre !\n");
    printf("L'air immense ouvre et referme mon livre,\n");
    printf("La vague en poudre ose jaillir des rocs !\n");
    printf("Envolez-vous, pages tout éblouies !\n");

    return 0;
}
```

Exercice 3 : L'école des erreurs

Le quotidien d'un développeur est rempli de messages d'erreur. Il est vital d'apprendre à les lire.

1. Reprenez le programme `hello.c` original.
2. Supprimez volontairement le point-virgule `;` à la fin de la ligne `printf`.
3. Essayez de le compiler. Lisez attentivement le message d'erreur. Le compilateur vous dira souvent la ligne exacte (ou la ligne juste après) où il a détecté un problème. Il pourrait dire quelque chose comme : `error: expected ';' before 'return'`.
4. Remettez le point-virgule, puis supprimez l'accolade fermante `}` à la fin du programme.
5. Recompilez et observez le nouveau message d'erreur.

Cet exercice a pour but de vous familiariser avec les messages d'erreur. Ne les craignez pas, ils sont vos meilleurs alliés pour trouver et corriger vos bugs.

Exemple de simulation d'erreur (point-virgule manquant) :

```c
#include <stdio.h>

int main() {
    // Erreur: le point-virgule est manquant a la fin de la ligne ci-dessous
    printf("Hello, World!\n")

    return 0;
}
```

Lors de la compilation de ce code (`gcc hello.c -o hello`), vous obtiendriez une erreur similaire à :

```
hello.c:5:5: error: expected ';' before 'return'
     return 0;
     ^~~~~~
```

Variables, Types et Opérateurs

Introduction : Qu'est-ce qu'une variable ?

En programmation, une variable est une zone de stockage nommée que votre programme peut manipuler. Imaginez des boîtes de rangement avec des étiquettes. Chaque boîte peut contenir quelque chose (un nombre, une lettre, etc.), et l'étiquette vous permet de savoir quelle boîte utiliser.

En C, chaque variable a un **type**, qui détermine la nature des données qu'elle peut contenir (par exemple, un nombre entier ou un caractère) et la quantité de mémoire à lui allouer.

Avant de pouvoir utiliser une variable, vous devez la **déclarer**, c'est-à-dire lui donner un type et un nom.

Les types de données de base

Le C propose plusieurs types de données fondamentaux. Voici les plus courants :

- `int` : Utilisé pour stocker des nombres **entiers** (sans partie décimale), comme -10, 0, ou 120.
- `float` : Pour les nombres à **virgule flottante** en simple précision. Utile pour les nombres avec une partie décimale, comme 3.14 ou -0.0025.

- `double` : Similaire à `float`, mais pour les nombres à virgule flottante en **double précision**. Il offre une plus grande précision et peut contenir des nombres plus grands, ce qui en fait le choix par défaut pour les calculs décimaux.
- `char` : Conçu pour stocker un **unique caractère**, comme 'a', 'B', ou '?'. En réalité, il stocke un petit nombre entier correspondant au code du caractère dans la table ASCII.

Il existe également des modificateurs comme `short`, `long`, `signed` (par défaut) et `unsigned` qui permettent d'ajuster la plage de valeurs que chaque type peut contenir.

Déclaration et Initialisation

La déclaration

Déclarer une variable signifie simplement annoncer son type et son nom au compilateur. La syntaxe est : `type nom_de_la_variable;`.

```
int nombreDeVies; // Declare une variable pour un entier
float score;       // Declare une variable pour un nombre a virgule
char touche;       // Declare une variable pour un caractere
```

Une fois déclarée, une variable contient une valeur "poubelle" (la donnée qui se trouvait en mémoire à cet endroit). Il est donc crucial de lui donner une valeur initiale.

L'initialisation

Initialiser une variable, c'est lui assigner sa première valeur. On le fait avec l'opérateur d'assignation `=`.

```
int nombreDeVies;
nombreDeVies = 5; // Initialisation

float score;
score = 120.5;

char touche;
touche = 'A';
```

On peut (et on devrait souvent) déclarer et initialiser une variable en une seule ligne :

```
int nombreDeVies = 5;
float score = 120.5;
char touche = 'A';
```

Les constantes

Si vous avez une variable dont la valeur ne doit jamais changer, vous pouvez la déclarer comme une **constante** avec le mot-clé `const`. Toute tentative de la modifier plus tard dans le code provoquera une erreur de compilation.

```
const float PI = 3.14159;
// PI = 3.14; // Ceci provoquera une erreur !
```

Les Opérateurs

Les opérateurs sont des symboles qui effectuent des opérations sur des variables et des valeurs.

Opérateurs arithmétiques

Ce sont les opérateurs mathématiques de base :

- `+` : Addition
- `-` : Soustraction
- `*` : Multiplication
- `/` : Division
- `%` : Modulo (donne le reste d'une division entière)

```
int a = 10;
int b = 3;

int somme = a + b; // 13
int diff = a - b;  // 7
int produit = a * b; // 30

// Attention a la division entiere !
int division = a / b; // 3 (la partie decimale est tronquee)

int reste = a % b; // 1 (car 10 = 3*3 + 1)
```

Pour une division avec un résultat décimal, au moins un des opérandes doit être de type `float` ou `double`.

Opérateurs de comparaison et logiques

Ils sont essentiels pour prendre des décisions dans le code.

- Comparaison : `==` (égal à), `!=` (différent de), `<`, `>`, `<=`, `>=`
- Logiques : `&&` (ET logique), `||` (OU logique), `!` (NON logique)

Opérateurs d'incrémentation et de décrémentation

Très courants, ils permettent d'ajouter ou de soustraire 1 à une variable.

- `++` : Incrémentation. `x++` est un raccourci pour `x = x + 1`.
- `--` : Décrémentation. `x--` est un raccourci pour `x = x - 1`.

Exercices

Exercice 1 : Calculateur de périmètre

Écrivez un programme qui déclare deux variables entières, `longueur` et `largeur`, avec les valeurs de votre choix. Calculez et affichez le périmètre du rectangle correspondant.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    int longueur = 10;
    int largeur = 5;
    int perimetre = 2 * (longueur + largeur);

    printf("Un rectangle de longueur %d et de largeur %d a un périmètre de %d.\n", longu

    return 0;
}
```

Exercice 2 : Conversion de température

Écrivez un programme qui convertit une température de degrés Celsius en degrés Fahrenheit. La formule est : F = C * 9/5 + 32.
Utilisez des variables de type `float` ou `double` pour un résultat précis.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    float celsius = 20.0;
    float fahrenheit = celsius * 9.0 / 5.0 + 32; // Utiliser 9.0/5.0 pour une division f

    printf("%.1f degrés Celsius équivalent à %.1f degrés Fahrenheit.\n", celsius, fahren

    return 0;
}
```

Exercice 3 : Pair ou Impair ?

Écrivez un programme qui déclare une variable entière et qui, en utilisant l'opérateur modulo, détermine si le nombre est pair ou impair.
Indice : Un nombre est pair si le reste de sa division par 2 est 0.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    int nombre = 7;

    if (nombre % 2 == 0) {
        printf("Le nombre %d est pair.\n", nombre);
    } else {
        printf("Le nombre %d est impair.\n", nombre);
    }

    return 0;
}
```

Entrées, Sorties et Structures de Contrôle

Jusqu'à présent, nos programmes étaient statiques : ils exécutaient les mêmes instructions dans le même ordre à chaque fois. Pour créer des applications vraiment utiles, nous devons pouvoir interagir avec l'utilisateur (entrées), afficher des résultats variables (sorties), et prendre des décisions ou répéter des actions en fonction des circonstances (structures de contrôle).

Les Sorties Formatées avec `printf`

Nous connaissons déjà `printf` pour afficher du texte simple. Sa vraie puissance réside dans sa capacité à "formater" des chaînes de caractères, c'est-à-dire à y insérer des valeurs de variables de manière structurée.

Pour cela, on utilise des **spécificateurs de format**. Ce sont des codes spéciaux commençant par `%` qui agissent comme des emplacements réservés pour les variables.

- `%d` ou `%i` : pour les nombres entiers (`int`).
- `%f` : pour les nombres à virgule flottante (`float` ou `double`).
- `%c` : pour un unique caractère (`char`).
- `%s` : pour les chaînes de caractères (que nous verrons plus tard).

Comment ça marche ?
On place les spécificateurs dans la chaîne de caractères, puis on fournit les variables correspondantes dans le même ordre, après la chaîne, séparées par des virgules.

```c
#include <stdio.h>

int main() {
    int age = 25;
    float poids = 72.5;
    char initial = 'L';

    printf("Bonjour, je m\'appelle %c.\n", initial);
    printf("J\'ai %d ans et je pèse %.1f kg.\n", age, poids);
    // Le ".1" dans "%.1f" demande d'afficher le float avec une seule décimale.

    return 0;
}
```

Résultat attendu :

```
Bonjour, je m\'appelle L.
J\'ai 25 ans et je pèse 72.5 kg.
```

Les Entrées Utilisateur avec `scanf`

Pour rendre un programme interactif, il faut pouvoir lire des données tapées par l'utilisateur au clavier. La fonction `scanf` (scan formatted) est la contrepartie de `printf`.

Elle lit une entrée depuis le terminal, essaie de la faire correspondre au format que vous spécifiez, et stocke la valeur lue dans une variable.

Attention : La syntaxe de `scanf` est un peu particulière. Pour qu'elle puisse modifier la valeur d'une variable, nous devons lui donner l'**adresse mémoire** de cette variable. On obtient cette adresse en préfixant le nom de la variable avec le symbole `&` (l'opérateur "adresse de").

```c
#include <stdio.h>

int main() {
    int ageUtilisateur;

    printf("Quel âge avez-vous ? ");
    scanf("%d", &ageUtilisateur); // On lit un entier et on le stocke à l'adresse de age

    printf("Ah, vous avez donc %d ans !\n", ageUtilisateur);

    return 0;
}
```

Déroulement de l'exécution :

1. Le programme affiche "Quel âge avez-vous ? ".
2. Il se met en pause et attend que l'utilisateur tape quelque chose et appuie sur Entrée.
3. Si l'utilisateur tape "30", `scanf` lit cette valeur, la convertit en entier et la place dans la variable `ageUtilisateur`.
4. Le programme reprend et affiche "Ah, vous avez donc 30 ans !".

Les Structures de Contrôle : Prendre des Décisions

Les structures de contrôle permettent de modifier le flux d'exécution du programme.

La condition `if , else if , else`

C'est la structure de décision la plus fondamentale. Elle permet d'exécuter un bloc de code uniquement si une certaine condition est vraie.

- `if (condition)` : Exécute le bloc qui suit si la `condition` est vraie.
- `else if (autre_condition)` : Si la première condition est fausse, teste cette nouvelle condition.
- `else` : Exécute le bloc qui suit si aucune des conditions précédentes n'était vraie.

```c
#include <stdio.h>

int main() {
    int note;
    printf("Entrez votre note (sur 20) : ");
    scanf("%d", &note);

    if (note >= 18) {
        printf("Excellent travail ! Mention Très Bien.\n");
    } else if (note >= 14) {
        printf("Bon travail. Mention Bien.\n");
    } else if (note >= 10) {
        printf("Admis. Mention Assez Bien.\n");
    } else {
        printf("Désolé, vous n\'êtes pas admis.\n");
    }

    return 0;
}
```

La structure `switch`

Quand vous avez une série de `if / else if` qui testent tous la même variable pour différentes valeurs, la structure `switch` peut rendre le code plus lisible.

Elle évalue une variable une seule fois, puis saute au `case` correspondant à la valeur. Le mot-clé `break;` est essentiel pour sortir du `switch` après avoir exécuté un `case` . Sans lui, le programme continuerait d'exécuter les `case` suivants !

```c
#include <stdio.h>

int main() {
    int choix;
    printf("Menu :\n1. Jouer\n2. Options\n3. Quitter\nVotre choix : ");
    scanf("%d", &choix);

    switch (choix) {
        case 1:
            printf("Lancement du jeu...\n");
            break; // Ne pas oublier !
        case 2:
            printf("Affichage des options...\n");
            break;
        case 3:
            printf("Au revoir !\n");
            break;
        default: // Le cas par défaut, si aucune valeur ne correspond
            printf("Choix invalide.\n");
            break;
    }

    return 0;
}
```

Les Structures de Contrôle : Répéter des Actions (Les Boucles)

Les boucles permettent d'exécuter un même bloc de code plusieurs fois.

La boucle `while`

La boucle `while` (tant que) répète un bloc de code **tant qu'une condition est vraie**. La condition est testée *avant* chaque tour de boucle. Si la condition est fausse dès le départ, la boucle ne s'exécute jamais.

```c
#include <stdio.h>

int main() {
    int i = 1; // 1. Initialisation

    while (i <= 5) { // 2. Condition
        printf("Ceci est le tour de boucle numéro %d.\n", i);
        i++; // 3. Itération (TRÈS IMPORTANT pour ne pas faire une boucle infinie)
    }

    printf("Fin de la boucle.\n");
    return 0;
}
```

La boucle `for`

La boucle `for` (pour) est idéale quand on sait à l'avance combien de fois on veut répéter une action. Elle regroupe l'initialisation, la condition et l'itération en une seule ligne, ce qui la rend très compacte et lisible.

La syntaxe est : `for (initialisation; condition; itération)`

```c
#include <stdio.h>

int main() {
    // La même chose que l'exemple while, mais en plus court !
    for (int i = 1; i <= 5; i++) {
        printf("Ceci est le tour de boucle numéro %d.\n", i);
    }

    printf("Fin de la boucle.\n");
    return 0;
}
```

Exercices

Exercice 1 : La calculatrice simple

Écrivez un programme qui demande à l'utilisateur de saisir deux nombres (`float`), puis de choisir une opération (1 pour addition, 2 pour soustraction, 3 pour multiplication, 4 for division). Le programme doit ensuite afficher le résultat de l'opération.
Utilisez `scanf` pour les nombres et le choix, et une structure `switch` pour effectuer le bon calcul.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    float a, b;
    int operation;

    printf("Entrez le premier nombre : ");
    scanf("%f", &a);
    printf("Entrez le second nombre : ");
    scanf("%f", &b);

    printf("\nChoisissez une opération :\n");
    printf("1. Addition\n");
    printf("2. Soustraction\n");
    printf("3. Multiplication\n");
    printf("4. Division\n");
    printf("Votre choix : ");
    scanf("%d", &operation);

    switch (operation) {
        case 1:
            printf("Résultat : %.2f + %.2f = %.2f\n", a, b, a + b);
            break;
        case 2:
            printf("Résultat : %.2f - %.2f = %.2f\n", a, b, a - b);
            break;
        case 3:
            printf("Résultat : %.2f * %.2f = %.2f\n", a, b, a * b);
            break;
        case 4:
            if (b != 0) {
                printf("Résultat : %.2f / %.2f = %.2f\n", a, b, a / b);
            } else {
                printf("Erreur : Division par zéro impossible.\n");
            }
            break;
        default:
            printf("Opération non valide.\n");
            break;
    }
```

```
    return 0;
}
```

Exercice 2 : Le jeu du "Plus ou Moins"

Écrivez un programme qui :

1. "Pense" à un nombre secret entre 1 et 100 (vous pouvez le définir en dur dans une constante pour l'instant, ex: `const int SECRET = 42;`).
2. Demande à l'utilisateur de deviner le nombre.
3. Utilise une boucle `while` pour continuer à demander un nombre tant que l'utilisateur n'a pas trouvé.
4. Après chaque tentative, le programme indique si le nombre secret est "plus grand" ou "plus petit".
5. Quand l'utilisateur trouve, le programme le félicite et s'arrête.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    const int NOMBRE_SECRET = 42;
    int proposition = 0;

    printf("J\'ai choisi un nombre entre 1 et 100. À vous de le deviner !\n");

    while (proposition != NOMBRE_SECRET) {
        printf("Votre proposition ? ");
        scanf("%d", &proposition);

        if (proposition < NOMBRE_SECRET) {
            printf("C\'est plus grand !\n");
        } else if (proposition > NOMBRE_SECRET) {
            printf("C\'est plus petit !\n");
        }
    }

    printf("Félicitations ! Le nombre secret était bien %d.\n", NOMBRE_SECRET);

    return 0;
}
```

Exercice 3 : La table de multiplication

Écrivez un programme qui demande à l'utilisateur de choisir un nombre entier. Le programme doit ensuite afficher la table de multiplication de ce nombre, de 1 à 10, en utilisant une boucle `for`.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    int nombre;
    printf("Choisissez un nombre pour voir sa table de multiplication : ");
    scanf("%d", &nombre);

    printf("--- Table de %d ---\n", nombre);
    for (int i = 1; i <= 10; i++) {
        printf("%d x %d = %d\n", nombre, i, nombre * i);
    }

    return 0;
}
```

Les Fonctions

À mesure que nos programmes grandissent, il devient essentiel de les organiser. Copier-coller du code est une mauvaise pratique : si vous devez corriger un bug, vous devrez le faire à plusieurs endroits. Les fonctions sont la solution à ce problème. Elles sont le principal outil d'organisation et de réutilisation du code en C.

Qu'est-ce qu'une fonction ?

Une fonction est un bloc de code autonome et nommé qui accomplit une tâche spécifique. Vous pouvez voir une fonction comme une "mini-machine" ou une "recette" :

- Elle peut prendre des ingrédients en entrée (les **paramètres**).
- Elle exécute une série d'instructions.
- Elle peut produire un résultat en sortie (la **valeur de retour**).

Nous utilisons des fonctions depuis le début : `printf`, `scanf` et même `main` sont des fonctions !

Pourquoi utiliser des fonctions ?

1. **Réutilisabilité :** Une fois une fonction écrite, vous pouvez l'**appeler** (l'exécuter) autant de fois que vous le souhaitez depuis n'importe où dans votre programme.
2. **Modularité :** Elles permettent de découper un gros problème complexe en plusieurs petits sous-problèmes plus simples à résoudre. Chaque fonction résout un de ces sous-problèmes.
3. **Lisibilité :** Un code bien découpé en fonctions est plus facile à lire et à comprendre. `calculerAireRectangle(10, 5);` est bien plus clair qu'un bloc de calculs au milieu de votre `main`.
4. **Maintenance :** Si vous devez modifier la manière dont une tâche est effectuée, vous n'avez qu'à modifier le code à un seul endroit : à l'intérieur de la fonction concernée.

Anatomie d'une fonction en C

Une fonction se compose de deux parties principales : la **déclaration** (ou prototype) et la **définition**.

1. La Définition de la fonction

C'est là que vous écrivez le code de la fonction, ce qu'elle fait réellement. Sa syntaxe est la suivante :

```
type_de_retour nom_de_la_fonction(type1 parametre1, type2 parametre2, ...)
{   // Corps de la fonction : instructions
    return valeur; // Si le type de retour n'est pas void
}
```

- **`type_de_retour`** : Le type de la valeur que la fonction renvoie (ex: `int`, `float`, `char`). Si la fonction ne renvoie rien, on utilise le mot-clé `void`.
- **`nom_de_la_fonction`** : Un nom descriptif que vous lui donnez (ex: `calculerPerimetre`, `afficherMenu`).
- **`paramètres`** : La liste des "ingrédients" dont la fonction a besoin pour travailler. Chaque paramètre a un type et un nom. S'il n'y en a pas, on laisse les parenthèses vides : `()`.
- **`Corps de la fonction`** : Le bloc de code entre accolades `{}` qui est exécuté lorsque la fonction est appelée.
- **`return`** : Le mot-clé qui termine la fonction et renvoie une valeur. La valeur doit être du même type que le `type_de_retour`.

Exemple : Une fonction qui additionne deux entiers.

```c
int addition(int a, int b) {
    int resultat = a + b;
    return resultat; // Renvoie la somme
}
```

2. L'Appel de la fonction

Pour exécuter une fonction, on l'appelle simplement par son nom, en lui fournissant les valeurs nécessaires (les **arguments**) entre parenthèses.

```c
#include <stdio.h>

// Définition de la fonction
int addition(int a, int b) {
    return a + b;
}

int main() {
    int x = 10;
    int y = 25;

    // Appel de la fonction 'addition' avec les arguments x et y
    // La valeur de retour est stockée dans la variable 'somme'
    int somme = addition(x, y);

    printf("La somme de %d et %d est %d.\n", x, y, somme); // Affiche "La somme de 10 et

    // On peut aussi utiliser directement le retour de la fonction
    printf("Une autre somme : %d\n", addition(100, 50)); // Affiche "Une autre somme : 1

    return 0;
}
```

3. La Déclaration (Prototype)

Que se passe-t-il si vous définissez votre fonction *après* la fonction `main` ? Le compilateur lira `main`, verra un appel à une fonction qu'il ne connaît pas encore, et générera une erreur.

Pour régler cela, on utilise une **déclaration de fonction**, aussi appelée **prototype**. C'est simplement la première ligne de la définition de la fonction (son "en-tête"), terminée par un point-virgule. On la

place généralement en haut du fichier, après les `#include` .

Le prototype informe le compilateur : "Ne t'inquiète pas, il existe une fonction avec ce nom, qui prend ces paramètres et renvoie ce type. Tu trouveras son code plus loin."

```c
#include <stdio.h>

// Prototype de la fonction 'addition'
int addition(int a, int b);

int main() {
    int somme = addition(10, 25); // Le compilateur sait que 'addition' existe grâce au
    printf("La somme est %d.\n", somme);
    return 0;
}

// Définition de la fonction (peut maintenant être après main)
int addition(int a, int b) {
    return a + b;
}
```

Passage par Valeur

C'est un concept **fondamental** en C. Lorsque vous passez une variable en argument à une fonction, la fonction ne reçoit pas la variable originale, mais une **copie** de sa valeur.

Cela signifie que si la fonction modifie la valeur de son paramètre, cela n'a **aucun effet** sur la variable originale dans la fonction appelante.

```c
#include <stdio.h>

void essayerDeModifier(int x) {
    printf("Dans la fonction, x vaut %d avant modification.\n", x);
    x = 100; // On modifie la COPIE
    printf("Dans la fonction, x vaut %d après modification.\n", x);
}

int main() {
    int monNombre = 10;
    printf("Dans main, monNombre vaut %d avant l'appel.\n", monNombre);

    essayerDeModifier(monNombre);

    printf("Dans main, monNombre vaut %d après l'appel.\n", monNombre); // Toujours 10

    return 0;
}
```

Résultat :

```
Dans main, monNombre vaut 10 avant l'appel.
Dans la fonction, x vaut 10 avant modification.
Dans la fonction, x vaut 100 après modification.
Dans main, monNombre vaut 10 après l'appel.
```

Exercices

Exercice 1 : Aire d'un rectangle

Écrivez une fonction `calculerAire` qui prend deux `float` (longueur et largeur) en paramètres et qui retourne leur produit (l'aire). Dans `main`, demandez à l'utilisateur de saisir une longueur et une largeur, appelez votre fonction, puis affichez le résultat.

Exemple de solution :

```c
#include <stdio.h>

// Prototype
float calculerAire(float longueur, float largeur);

int main() {
    float l, w;
    printf("Entrez la longueur du rectangle : ");
    scanf("%f", &l);
    printf("Entrez la largeur du rectangle : ");
    scanf("%f", &w);

    float aire = calculerAire(l, w);

    printf("L'aire d'un rectangle de %.2f x %.2f est de %.2f.\n", l, w, aire);

    return 0;
}

// Définition
float calculerAire(float longueur, float largeur) {
    return longueur * largeur;
}
```

Exercice 2 : Le plus grand des deux

Écrivez une fonction `max` qui prend deux entiers en paramètres et retourne le plus grand des deux. Dans `main`, testez votre fonction avec différentes valeurs et affichez le résultat.

Exemple de solution :

```c
#include <stdio.h>

int max(int num1, int num2);

int main() {
    int a = 15;
    int b = 32;

    printf("Le plus grand nombre entre %d et %d est %d.\n", a, b, max(a, b));
    printf("Le plus grand nombre entre %d et %d est %d.\n", 100, 50, max(100, 50));

    return 0;
}

int max(int num1, int num2) {
    if (num1 > num2) {
        return num1;
    } else {
        return num2;
    }
}
```

Exercice 3 : Afficher un menu

Écrivez une fonction `afficherMenu` qui ne prend aucun paramètre et ne retourne rien (`void`). Son seul rôle est d'afficher les lignes d'un menu pour une application (par exemple, les choix d'une calculatrice ou d'un jeu). Écrivez également une fonction `demanderChoix` qui ne prend aucun paramètre mais qui retourne un `int` . Elle demande à l'utilisateur de faire un choix et retourne la valeur saisie. Dans `main` , appelez `afficherMenu` puis `demanderChoix` et affichez le choix de l'utilisateur.

Exemple de solution :

```c
#include <stdio.h>

void afficherMenu(void); // (void) est optionnel mais indique clairement l'absence de pa
int demanderChoix(void);

int main() {
    afficherMenu();
    int choix = demanderChoix();

    printf("Vous avez choisi l'option %d.\n", choix);

    return 0;
}

void afficherMenu(void) {
    printf("--- MENU PRINCIPAL ---\n");
    printf("1. Démarrer une nouvelle partie\n");
    printf("2. Charger une partie\n");
    printf("3. Options\n");
    printf("4. Quitter\n");
    printf("----------------------\n");
}

int demanderChoix(void) {
    int c;
    printf("Votre choix : ");
    scanf("%d", &c);
    return c;
}
```

Les Tableaux

Jusqu'ici, chaque variable que nous avons créée ne pouvait contenir qu'une seule valeur à la fois (`int a = 5;`). Mais que faire si nous devons stocker une liste de valeurs, comme les 10 meilleures notes d'un examen ou les 7 températures de la semaine ? Déclarer `note1` , `note2` , `note3` ... serait extrêmement répétitif et peu pratique.

La solution est le **tableau**. Un tableau est une structure de données qui permet de stocker une collection de plusieurs éléments **de même type** dans une seule variable.

Déclaration d'un tableau

Pour déclarer un tableau, vous devez spécifier le type de ses éléments, son nom, et le nombre d'éléments qu'il peut contenir (sa taille) entre crochets `[]` .

```
type nom_du_tableau[TAILLE];
```

```c
// Un tableau pour stocker 5 notes entières
int notes[5];

// Un tableau pour stocker les 7 températures de la semaine
float temperatures[7];

// Un tableau pour stocker les 26 lettres de l'alphabet
char alphabet[26];
```

Lorsque vous déclarez un tableau de taille `N` , le compilateur réserve en mémoire un bloc contigu suffisamment grand pour stocker `N` éléments du type spécifié.

Accès aux éléments d'un tableau

Pour accéder à un élément spécifique du tableau, on utilise son **indice** (sa position).

Attention : En C, les indices commencent **toujours à 0**.

- Le premier élément a l'indice `0` .
- Le deuxième élément a l'indice `1` .
- ...
- Le dernier élément d'un tableau de taille `N` a l'indice `N - 1` .

On accède à un élément avec la syntaxe `nom_du_tableau[indice]` .

```c
int notes[5]; // Peut contenir 5 entiers aux indices 0, 1, 2, 3, 4

// Assigner des valeurs aux éléments du tableau
notes[0] = 15; // Premier élément
notes[1] = 12;
notes[2] = 18;
notes[3] = 9;
notes[4] = 14; // Dernier élément

// Lire et afficher la valeur d'un élément
printf("La troisième note est : %d\n", notes[2]); // Affiche 18

// Modifier une valeur existante
notes[0] = 16;
```

Tenter d'accéder à un indice en dehors des limites (par exemple `notes[5]` ou `notes[-1]`) est une erreur grave appelée **dépassement de tampon (buffer overflow)**. Le C ne vous en empêchera pas, mais cela peut corrompre la mémoire et faire planter votre programme (ou pire, créer une faille de sécurité).

Initialisation d'un tableau

Comme pour les variables simples, il est préférable d'initialiser un tableau dès sa déclaration pour éviter qu'il ne contienne des valeurs "poubelle". On le fait en fournissant une liste de valeurs entre accolades `{}` .

```c
// Déclaration et initialisation en une ligne
int notes[5] = {15, 12, 18, 9, 14};

// Si vous fournissez les valeurs, vous pouvez laisser le compilateur
// deviner la taille du tableau. C'est très pratique.
int notes[] = {15, 12, 18, 9, 14}; // Le compilateur comprend qu'il faut une taille de 5
```

Si vous initialisez un tableau avec moins de valeurs que sa taille, les éléments restants sont automatiquement initialisés à `0` .

```c
// Ce tableau de 10 éléments contiendra {5, 10, 15, 0, 0, 0, 0, 0, 0, 0}
int data[10] = {5, 10, 15};
```

Parcourir un tableau avec une boucle

La véritable puissance des tableaux apparaît lorsqu'on les combine avec des boucles, notamment la boucle `for`, pour effectuer une opération sur chaque élément.

Exemple : Calculer la moyenne de nos notes.

```c
#include <stdio.h>

int main() {
    int notes[] = {15, 12, 18, 9, 14};
    int somme = 0;
    // Astuce : On calcule la taille dynamiquement !
    int taille = sizeof(notes) / sizeof(notes[0]);

    // 1. Parcourir le tableau pour additionner toutes les notes
    for (int i = 0; i < taille; i++) {
        printf("Lecture de la note à l'indice %d : %d\n", i, notes[i]);
        somme = somme + notes[i]; // ou somme += notes[i];
    }

    // 2. Calculer la moyenne (attention à la division)
    float moyenne = (float)somme / taille;

    printf("La somme des notes est : %d\n", somme);
    printf("La moyenne des notes est : %.2f\n", moyenne);

    return 0;
}
```

Une astuce essentielle : `sizeof(tableau) / sizeof(tableau[0])`

Dans l'exemple ci-dessus, nous avons calculé la taille du tableau avec `int taille = sizeof(notes) / sizeof(notes[0]);`. C'est une technique très courante et importante en C.

- `sizeof(notes)` retourne la taille totale en octets de tout le tableau. Par exemple, si un `int` occupe 4 octets et que le tableau a 5 éléments, `sizeof(notes)` vaudra 20.
- `sizeof(notes[0])` retourne la taille en octets d'un seul élément du tableau (ici, un `int`, donc 4).

- En divisant la taille totale par la taille d'un élément (20 / 4), on obtient le nombre d'éléments dans le tableau (5).

Cette méthode est bien plus robuste que de coder la taille en dur, car si vous ajoutez des éléments au tableau lors de son initialisation, le calcul de la taille s'adaptera automatiquement !

Les tableaux multidimensionnels

Le C permet de créer des tableaux à plusieurs dimensions. Un tableau à deux dimensions peut être vu comme une grille ou un tableau de tableaux. C'est utile pour représenter des plateaux de jeu, des images, des matrices, etc.

```
type nom_du_tableau[NOMBRE_DE_LIGNES][NOMBRE_DE_COLONNES];
```

```c
// Un "plateau" de 3 lignes et 4 colonnes
int grille[3][4];

// Initialisation
int grille[3][4] = {
    {1, 2, 3, 4},   // Ligne 0
    {5, 6, 7, 8},   // Ligne 1
    {9, 10, 11, 12} // Ligne 2
};

// Accès à un élément : grille[ligne][colonne]
printf("L'élément à la ligne 1, colonne 2 est : %d\n", grille[1][2]); // Affiche 7

// Parcours avec des boucles imbriquées
for (int i = 0; i < 3; i++) { // Boucle pour les lignes
    for (int j = 0; j < 4; j++) { // Boucle pour les colonnes
        printf("%d ", grille[i][j]);
    }
    printf("\n"); // Nouvelle ligne après chaque ligne de la grille
}
```

Exercices

Exercice 1 : Le plus grand et le plus petit

Écrivez un programme qui :

1. Initialise un tableau d'entiers avec au moins 5 valeurs de votre choix.
2. Parcourt le tableau pour trouver la valeur maximale et la valeur minimale.
3. Affiche ces deux valeurs à la fin.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    int nombres[] = {45, 12, 89, 23, 5, 67};
    int taille = 6;

    // On initialise min et max avec la première valeur du tableau
    int min = nombres[0];
    int max = nombres[0];

    // On commence la boucle à 1, car l'élément 0 est déjà traité
    for (int i = 1; i < taille; i++) {
        if (nombres[i] < min) {
            min = nombres[i]; // Nouveau minimum trouvé
        }
        if (nombres[i] > max) {
            max = nombres[i]; // Nouveau maximum trouvé
        }
    }

    printf("Le plus petit nombre du tableau est : %d\n", min);
    printf("Le plus grand nombre du tableau est : %d\n", max);

    return 0;
}
```

Exercice 2 : Inverser un tableau

Écrivez un programme qui :

1. Demande à l'utilisateur de saisir 5 entiers et les stocke dans un tableau.
2. Affiche le tableau dans l'ordre original.
3. Affiche le tableau dans l'ordre inverse.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    int valeurs[5];
    int taille = 5;

    // 1. Lecture des valeurs
    printf("Veuillez saisir %d nombres entiers :\n", taille);
    for (int i = 0; i < taille; i++) {
        printf("Nombre %d : ", i + 1);
        scanf("%d", &valeurs[i]);
    }

    // 2. Affichage original
    printf("\nTableau original : ");
    for (int i = 0; i < taille; i++) {
        printf("%d ", valeurs[i]);
    }

    // 3. Affichage inversé
    printf("\nTableau inversé : ");
    // On part de la fin (taille - 1) et on décrémente jusqu'à 0
    for (int i = taille - 1; i >= 0; i--) {
        printf("%d ", valeurs[i]);
    }
    printf("\n");

    return 0;
}
```

Exercice 3 : Somme de deux tableaux

Écrivez un programme qui :

1. Définit deux tableaux d'entiers de même taille (par exemple, taille 4).
2. Crée un troisième tableau de la même taille.
3. Remplit le troisième tableau en calculant, pour chaque indice, la somme des éléments des deux premiers tableaux. (ex: `tab3[i] = tab1[i] + tab2[i]`).
4. Affiche le contenu des trois tableaux pour vérifier le résultat.

Exemple de solution :

```c
#include <stdio.h>

void afficherTableau(int tab[], int taille) {
    for (int i = 0; i < taille; i++) {
        printf("%d ", tab[i]);
    }
    printf("\n");
}

int main() {
    int tab1[] = {10, 20, 30, 40};
    int tab2[] = {5, 8, 2, 10};
    int taille = 4;
    int tab_somme[taille];

    // Calcul de la somme
    for (int i = 0; i < taille; i++) {
        tab_somme[i] = tab1[i] + tab2[i];
    }

    printf("Tableau 1 : ");
    afficherTableau(tab1, taille);

    printf("Tableau 2 : ");
    afficherTableau(tab2, taille);

    printf("Tableau Somme : ");
    afficherTableau(tab_somme, taille);

    return 0;
}
```

Les Pointeurs

C'est l'un des chapitres les plus importants, et souvent considéré comme le plus complexe, du langage C. Les pointeurs sont un outil extraordinairement puissant qui donne au C sa flexibilité et ses performances. Ils permettent une gestion manuelle et précise de la mémoire de l'ordinateur.

Qu'est-ce qu'un pointeur ?

Pour comprendre les pointeurs, il faut d'abord comprendre comment les variables sont stockées en mémoire. Chaque variable que vous déclarez (`int a` , `char c` , etc.) est stockée quelque part dans la mémoire vive (RAM) de votre ordinateur. Cet "endroit" a une **adresse**, un numéro unique qui permet au processeur de le retrouver, un peu comme une adresse postale pour une maison.

Un **pointeur** n'est rien de plus qu'une variable spéciale dont la valeur n'est pas un nombre ou un caractère, mais l'**adresse mémoire d'une autre variable**.

Au lieu de contenir une donnée, un pointeur "pointe vers" l'endroit où se trouve la donnée.

Opérateurs clés : `&` et `*`

Pour travailler avec les pointeurs, deux opérateurs sont absolument essentiels :

1. L'opérateur d'adresse : `&` (address-of)

Placé devant le nom d'une variable, l'opérateur `&` ne retourne pas sa valeur, mais son **adresse en mémoire**.

```
int age = 30;
// &age nous donne l'adresse où la valeur 30 est stockée.
printf("La valeur de 'age' est %d.\n", age);
printf("L'adresse mémoire de 'age' est %p.\n", &age); // %p est le spécificateur pour af
```

Le résultat de l'adresse sera un grand nombre hexadécimal, par exemple `0x7ffc9c7b4a54` *.*

2. L'opérateur de déréférencement : `*` (indirection)

Placé devant une variable de type pointeur, l'opérateur `*` permet d'accéder à la **valeur qui se trouve à l'adresse contenue dans le pointeur**. C'est l'opération inverse de `&` . On "suit" le pointeur pour voir ce qu'il y a au bout.

Déclaration et Utilisation des Pointeurs

Déclaration

On déclare un pointeur en spécifiant le type de données vers lequel il va pointer, suivi d'un astérisque `*` et du nom du pointeur.

```
type_de_donnee *nom_du_pointeur;
```

```
int *ptr_sur_entier;   // Un pointeur capable de pointer vers une variable de type int
float *ptr_sur_float; // Un pointeur capable de pointer vers une variable de type float
char *ptr_sur_char;   // Un pointeur capable de pointer vers une variable de type char
```

Le type est important : il indique au compilateur combien d'octets lire en mémoire lorsqu'on déréférence le pointeur.

Assignation et Déréférencement

Voici un cycle complet :

```c
#include <stdio.h>

int main() {
    int age = 30;          // 1. Une variable simple
    int *ptr_age = NULL;   // 2. On déclare un pointeur sur un entier.
                           //    Il est bon de l'initialiser à NULL.

    ptr_age = &age;        // 3. On assigne l'adresse de 'age' au pointeur 'ptr_age'.
                           //    Maintenant, ptr_age "pointe vers" age.

    printf("Valeur de age : %d\n", age);
    printf("Adresse de age : %p\n", &age);
    printf("Valeur de ptr_age (l'adresse qu'il contient) : %p\n", ptr_age);

    // 4. On utilise l'opérateur * pour accéder à la valeur pointée
    printf("Valeur pointée par ptr_age : %d\n", *ptr_age);

    // On peut aussi modifier la variable originale via le pointeur
    *ptr_age = 31; // "Va à l'adresse contenue dans ptr_age et mets-y la valeur 31"

    printf("La nouvelle valeur de age est : %d\n", age); // Affiche 31 !

    return 0;
}
```

Le pointeur `NULL`

Un pointeur qui ne pointe vers rien de valide doit être initialisé à `NULL`. C'est une valeur spéciale (souvent équivalente à 0) qui signifie "ce pointeur ne pointe nulle part". Tenter de déréférencer un pointeur `NULL` provoquera un plantage du programme (une "segmentation fault"), ce qui est une bonne chose car cela signale immédiatement une erreur.

Pointeurs et Fonctions : Le Passage par Référence

Vous vous souvenez du "passage par valeur" ? Les fonctions reçoivent une copie des arguments, et ne peuvent donc pas modifier les variables originales. Les pointeurs résolvent ce problème.

En passant un pointeur (c'est-à-dire l'adresse d'une variable) à une fonction, on lui donne les "coordonnées" de la variable originale. La fonction peut alors utiliser cette adresse pour modifier directement la valeur à cet endroit. C'est ce qu'on appelle une simulation de **passage par référence**.

Exemple : Une fonction qui échange les valeurs de deux variables.

```c
#include <stdio.h>

// La fonction reçoit les adresses de a et b
void echanger(int *ptr_a, int *ptr_b) {
    int temp = *ptr_a;  // On stocke la valeur de a
    *ptr_a = *ptr_b;    // On met la valeur de b dans a
    *ptr_b = temp;      // On met l'ancienne valeur de a dans b
}

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

    printf("Avant échange : x = %d, y = %d\n", x, y);

    // On passe les adresses de x et y à la fonction
    echanger(&x, &y);

    printf("Après échange : x = %d, y = %d\n", x, y); // Affiche x = 20, y = 10

    return 0;
}
```

Exercices

Exercice 1 : Exploration de base

Écrivez un programme qui :

1. Déclare une variable `float` nommée `valeur` et l'initialise à `123.45`.
2. Déclare un pointeur sur `float` nommé `ptr_valeur`.
3. Fait pointer `ptr_valeur` sur `valeur`.
4. Affiche la valeur de `valeur` de deux manières : en utilisant la variable elle-même, puis en déréférençant le pointeur.
5. Affiche l'adresse de `valeur` de deux manières : en utilisant l'opérateur `&` sur la variable, puis en affichant le contenu du pointeur.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    float valeur = 123.45;
    float *ptr_valeur = &valeur;

    printf("--- Affichage de la valeur ---\n");
    printf("Via la variable : %.2f\n", valeur);
    printf("Via le pointeur : %.2f\n", *ptr_valeur);

    printf("\n--- Affichage de l'adresse ---\n");
    printf("Via l'opérateur & : %p\n", &valeur);
    printf("Via le pointeur   : %p\n", ptr_valeur);

    return 0;
}
```

Exercice 2 : Incrémentation par pointeur

Écrivez une fonction `incrementer` qui prend un pointeur sur un entier en paramètre. La fonction ne retourne rien (`void`), mais elle doit incrémenter de 1 la valeur de la variable pointée. Dans `main` , déclarez un entier, appelez la fonction pour le modifier, et vérifiez que sa valeur a bien changé.

Exemple de solution :

```c
#include <stdio.h>

void incrementer(int *nombre) {
    // On accède à la valeur pointée et on l'incrémente
    (*nombre)++;
    // Les parenthèses sont importantes à cause de la priorité des opérateurs.
    // *nombre++ serait interprété différemment (incrémenter le pointeur).
}

int main() {
    int compteur = 5;
    printf("La valeur du compteur avant l'appel : %d\n", compteur);

    incrementer(&compteur);

    printf("La valeur du compteur après l'appel : %d\n", compteur);

    return 0;
}
```

Exercice 3 : Calculs via pointeurs

Écrivez une fonction `calculer` qui prend deux pointeurs sur des entiers (`ptr_a` , `ptr_b`) et un pointeur sur un troisième entier (`ptr_somme`). La fonction doit calculer la somme des valeurs pointées par `ptr_a` et `ptr_b` et stocker le résultat à l'adresse pointée par `ptr_somme` .

Exemple de solution :

```c
#include <stdio.h>

void calculer(int *ptr_a, int *ptr_b, int *ptr_somme) {
    *ptr_somme = *ptr_a + *ptr_b;
}

int main() {
    int num1 = 50;
    int num2 = 30;
    int resultat; // Pas besoin d'initialiser, la fonction va le faire

    // On passe les adresses des trois variables
    calculer(&num1, &num2, &resultat);

    printf("La somme de %d et %d est %d.\n", num1, num2, resultat);

    return 0;
}
```

Les Chaînes de Caractères

En C, il n'existe pas de type de données de base "chaîne de caractères" (ou "string") comme en Python ou Java. À la place, une chaîne de caractères est une **convention** : c'est un **tableau de `char`** qui se termine par un caractère spécial, le **caractère nul `\0`** .

Ce caractère `\0` (null terminator) est crucial. Il agit comme un marqueur de fin, indiquant aux fonctions où la chaîne se termine.

Déclaration et Initialisation

Il y a deux manières principales de créer une chaîne de caractères.

1. Avec un littéral de chaîne (la méthode simple)

C'est la méthode la plus courante. Vous utilisez les guillemets doubles, et le compilateur s'occupe de tout, y compris d'ajouter le `\0` final.

```c
char salutation[] = "Bonjour";
```

En mémoire, cela crée un tableau de 8 `char` :
`{'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'}`

2. Avec un tableau de caractères (la méthode manuelle)

Vous pouvez aussi l'initialiser comme n'importe quel tableau. Si vous le faites, **n'oubliez pas d'ajouter `\0` vous-même !**

```
char salutation[8] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
```

Afficher et Lire des Chaînes

On utilise les spécificateurs de format `%s` dans `printf` et `scanf` .

```
#include <stdio.h>

int main() {
    char nom[50]; // On prévoit un tableau assez grand pour le nom

    printf("Quel est votre nom ? ");
    scanf("%s", nom); // Pas besoin de '&' pour les chaînes avec scanf !

    printf("Bonjour, %s !\n", nom);

    return 0;
}
```

Attention : `scanf` est dangereux !
`scanf("%s", ...)` s'arrête au premier espace et ne vérifie pas la taille du tableau. Si l'utilisateur tape un nom plus long que 49 caractères, cela provoquera un **dépassement de tampon (buffer overflow)**.

Une alternative plus sûre : `fgets`

La fonction `fgets` est bien meilleure car elle permet de spécifier la taille maximale à lire.

`fgets(destination, taille, source);`

- `destination` : Le tableau où stocker la chaîne.
- `taille` : La taille maximale à lire (incluant la place pour `\0`).

- `source` : D'où lire les données. Pour le clavier, on utilise `stdin`.

```c
#include <stdio.h>
#include <string.h> // Nécessaire pour strcspn

int main() {
    char nomComplet[50];

    printf("Quel est votre nom complet ? ");
    fgets(nomComplet, 50, stdin);

    // Astuce : On supprime le '\n' final capturé par fgets.
    nomComplet[strcspn(nomComplet, "\n")] = '\0';

    printf("Ravi de vous rencontrer, %s !\n", nomComplet);

    return 0;
}
```

Comment gérer le retour à la ligne de `fgets` ?

Un petit défaut de `fgets` est qu'il stocke le caractère de nouvelle ligne (`\n`) dans votre chaîne si l'utilisateur appuie sur Entrée avant d'avoir rempli le tampon. L'exemple ci-dessus montre la méthode la plus propre pour s'en débarrasser :

- `strcspn(nomComplet, "\n")` : Cette fonction de `<string.h>` recherche dans `nomComplet` le premier caractère qui est aussi dans la chaîne `"\n"`. Elle retourne l'indice de ce caractère.
- `nomComplet[...] = '\0';` : On utilise cet indice pour placer le caractère nul `\0` juste à la place du `\n`, coupant ainsi la chaîne juste avant.

La Bibliothèque `<string.h>`

Le C fournit une bibliothèque standard, `string.h`, remplie de fonctions utiles pour manipuler les chaînes de caractères. Pour l'utiliser, il faut ajouter `#include <string.h>` en haut de votre fichier. Une autre bibliothèque très utile dans ce contexte est `<ctype.h>`, qui offre des fonctions pour tester et manipuler les caractères (comme les mettre en majuscule ou minuscule).

Voici quelques fonctions incontournables :

La bibliothèque `<ctype.h>` : Manipuler les caractères

Avant de voir les fonctions de `<string.h>`, il est utile de connaître `<ctype.h>`. Elle est parfaite pour traiter les caractères un par un.

- `tolower(char c)` : Convertit un caractère majuscule en minuscule.
- `toupper(char c)` : Convertit un caractère minuscule en majuscule.
- `isdigit(char c)` : Vérifie si un caractère est un chiffre (0-9).
- `isalpha(char c)` : Vérifie si un caractère est une lettre.

La bibliothèque `<string.h>` : Manipuler les chaînes

`strlen` : Calculer la longueur

`strlen(chaine)` retourne la longueur d'une chaîne, **sans compter le caractère nul `\0`**.

```c
char msg[] = "Hello";
int longueur = strlen(msg); // longueur vaut 5
```

`strcpy` : Copier une chaîne

`strcpy(destination, source)` copie le contenu de la chaîne `source` dans la chaîne `destination`.
Attention : La destination doit être assez grande pour accueillir la source !

```c
char original[] = "Texte original";
char copie[20]; // Assez de place

strcpy(copie, original);
// Maintenant, 'copie' contient "Texte original"
```

`strcat` : Concaténer des chaînes

`strcat(destination, source)` ajoute (concatène) la chaîne `source` à la fin de la chaîne `destination`.
Attention : La destination doit être assez grande pour contenir les deux chaînes réunies !

```c
char debut[30] = "Bonjour ";
char fin[] = "le monde !";

strcat(debut, fin);
// Maintenant, 'debut' contient "Bonjour le monde !"
```

`strcmp` : Comparer des chaînes

`strcmp(chaine1, chaine2)` compare deux chaînes lexicographiquement (selon l'ordre du dictionnaire). On ne peut **pas** utiliser `==` pour comparer des chaînes.
Elle retourne :

- `0` si les chaînes sont identiques.
- une valeur `< 0` si `chaine1` vient avant `chaine2` .
- une valeur `> 0` si `chaine1` vient après `chaine2` .

```c
if (strcmp(motDePasse, "secret123") == 0) {
    printf("Accès autorisé.\n");
} else {
    printf("Accès refusé.\n");
}
```

Exercices

Exercice 1 : Compteur de voyelles

Écrivez un programme qui demande à l'utilisateur de saisir une phrase, puis qui compte et affiche le nombre de voyelles ('a', 'e', 'i', 'o', 'u', 'y', en minuscules et majuscules).

Exemple de solution :

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h> // Pour la fonction tolower

int main() {
    char phrase[256];
    int compteur = 0;

    printf("Entrez une phrase : ");
    fgets(phrase, 256, stdin);

    for (int i = 0; i < strlen(phrase); i++) {
        char lettre = tolower(phrase[i]); // On convertit en minuscule pour simplifier
        if (lettre == 'a' || lettre == 'e' || lettre == 'i' || lettre == 'o' || lettre =
            compteur++;
        }
    }

    printf("Cette phrase contient %d voyelles.\n", compteur);

    return 0;
}
```

Exercice 2 : Palindrome

Un palindrome est un mot ou une phrase qui se lit de la même manière dans les deux sens (ex: "radar", "level"). Écrivez un programme qui demande un mot à l'utilisateur et lui dit s'il s'agit d'un palindrome.
Indice : Comparez le premier caractère avec le dernier, le deuxième avec l'avant-dernier, et ainsi de suite.

Exemple de solution :

```c
#include <stdio.h>
#include <string.h>

int main() {
    char mot[100];
    int estPalindrome = 1; // On suppose que c'est vrai au départ

    printf("Entrez un mot pour vérifier si c'est un palindrome : ");
    scanf("%s", mot);

    int longueur = strlen(mot);
    for (int i = 0; i < longueur / 2; i++) {
        // On compare le caractère i avec le caractère symétrique à la fin
        if (mot[i] != mot[longueur - 1 - i]) {
            estPalindrome = 0; // Ce n'est pas un palindrome
            break; // Inutile de continuer la boucle
        }
    }

    if (estPalindrome) {
        printf("Le mot \"%s\" est un palindrome.\n", mot);
    } else {
        printf("Le mot \"%s\" n'est pas un palindrome.\n", mot);
    }

    return 0;
}
```

Exercice 3 : Création d'un nom d'utilisateur

Écrivez un programme qui demande à l'utilisateur son prénom et son nom de famille. Le programme doit ensuite créer un nom d'utilisateur en prenant la première lettre du prénom et en la concaténant avec le nom de famille (le tout en minuscules).
Exemple : Prénom "Loic", Nom "Dupont" -> Nom d'utilisateur "ldupont".

Exemple de solution :

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main() {
    char prenom[50];
    char nom[50];
    char username[101]; // Assez de place pour le résultat

    printf("Entrez votre prénom : ");
    scanf("%s", prenom);
    printf("Entrez votre nom de famille : ");
    scanf("%s", nom);

    // 1. On prend la première lettre du prénom
    username[0] = tolower(prenom[0]);
    username[1] = '\0'; // On transforme le caractère en une chaîne valide

    // 2. On convertit le nom de famille en minuscules
    for (int i = 0; i < strlen(nom); i++) {
        nom[i] = tolower(nom[i]);
    }

    // 3. On concatène
    strcat(username, nom);

    printf("Votre nom d'utilisateur suggéré est : %s\n", username);

    return 0;
}
```

Les Fichiers

Jusqu'à présent, toutes les données que nos programmes manipulaient étaient volatiles : elles disparaissaient dès que le programme se terminait. Pour conserver des informations de manière permanente, nous devons les stocker dans des **fichiers** sur le disque dur. Le C offre un ensemble de fonctions puissantes pour créer, lire et écrire dans des fichiers.

Le Pointeur de Fichier : `FILE`

En C, on ne manipule pas un fichier directement, mais à travers un **flux** (stream). Un flux est un canal de communication abstrait entre votre programme et le fichier. Pour gérer ce flux, on utilise un type spécial de pointeur : `FILE *` .

Toutes les opérations sur les fichiers commenceront par la création d'une variable de ce type.

```
FILE *fichier = NULL; // Déclaration d'un pointeur de fichier, initialisé à NULL
```

1. Ouvrir un Fichier : `fopen()`

Avant de pouvoir lire ou écrire, il faut ouvrir le fichier. La fonction `fopen()` s'en charge.

```
FILE *fopen(const char *nom_du_fichier, const char *mode);
```

- `nom_du_fichier` : Une chaîne de caractères contenant le chemin vers le fichier (ex: `"donnees.txt"`).
- `mode` : Une chaîne de caractères qui spécifie ce que vous voulez faire avec le fichier.

Les modes d'ouverture les plus courants :

- `"r"` (read) : Ouvre un fichier en **lecture seule**. Le fichier doit déjà exister.
- `"w"` (write) : Ouvre un fichier en **écriture**. Si le fichier existe, son contenu est **écrasé**. S'il n'existe pas, il est **créé**.
- `"a"` (append) : Ouvre un fichier en **écriture à la fin**. Si le fichier existe, les nouvelles données sont ajoutées après le contenu existant. S'il n'existe pas, il est créé.

Gestion des erreurs :
`fopen()` retourne un pointeur `FILE *` si l'ouverture réussit. Si une erreur se produit (fichier non trouvé en mode "r", pas les droits d'écriture, etc.), elle retourne `NULL` . **Il est absolument crucial de toujours tester cette valeur de retour.**

```c
FILE *fichier = fopen("scores.txt", "r");

if (fichier == NULL) {
    printf("Erreur : Impossible d'ouvrir le fichier scores.txt\n");
    return 1; // On quitte le programme avec un code d'erreur
}
// Si on arrive ici, le fichier est ouvert et prêt à être lu.
```

2. Fermer un Fichier : `fclose()`

Une fois que vous avez terminé de travailler avec un fichier, vous devez impérativement le fermer en utilisant `fclose()`.

`int fclose(FILE *pointeur_de_fichier);

Pourquoi est-ce si important ?

- **Sauvegarde des données :** Les données que vous écrivez sont souvent mises dans un tampon temporaire. `fclose()` s'assure que tout le contenu du tampon est bien écrit physiquement sur le disque.
- **Libération des ressources :** Chaque fichier ouvert consomme des ressources système. `fclose()` les libère.
- **Prévention de la corruption :** Laisser des fichiers ouverts peut entraîner une corruption des données si le programme se termine anormalement.

```c
fclose(fichier);
fichier = NULL; // Bonne pratique pour éviter d'utiliser un pointeur invalide
```

3. Écrire dans un Fichier

Il existe plusieurs fonctions pour écrire, mais la plus simple à apprendre est `fprintf()`, qui est la version "fichier" de `printf()`.

`int fprintf(FILE *flux, const char *format, ...);

Elle fonctionne exactement comme `printf`, mais prend en premier argument le pointeur vers le fichier dans lequel écrire.

```c
#include <stdio.h>

int main() {
    FILE *log = fopen("journal.log", "a"); // On ouvre en mode ajout

    if (log == NULL) {
        printf("Impossible de créer le fichier de log.\n");
        return 1;
    }

    fprintf(log, "Le programme a démarré avec succès.\n");
    int score = 150;
    fprintf(log, "L'utilisateur a atteint le score de %d.\n", score);

    fclose(log);
    printf("Les informations ont été enregistrées dans journal.log\n");

    return 0;
}
```

4. Lire depuis un Fichier

De même, il existe plusieurs façons de lire. La plus sûre et la plus flexible est `fgets()`, que nous connaissons déjà.

`char *fgets(char *destination, int taille, FILE *flux);

Elle lit une ligne entière depuis le fichier (jusqu'au `\n` ou jusqu'à `taille - 1` caractères) et la stocke dans `destination`. Elle retourne `NULL` quand elle atteint la fin du fichier.

Exemple : Lire un fichier ligne par ligne

```c
#include <stdio.h>

int main() {
    FILE *fichier = fopen("journal.log", "r");
    char ligne; // Un tampon pour stocker chaque ligne lue

    if (fichier == NULL) {
        printf("Impossible de lire le fichier journal.log\n");
        return 1;
    }

    printf("--- Contenu de journal.log ---\n");
    // La boucle continue tant que fgets ne retourne pas NULL (fin du fichier)
    while (fgets(ligne, sizeof(ligne), fichier) != NULL) {
        printf("%s", ligne); // On affiche la ligne lue
    }
    printf("--- Fin du fichier ---\n");

    fclose(fichier);
    return 0;
}
```

Exercices

Exercice 1 : Mon carnet de notes

Écrivez un programme qui demande à l'utilisateur de saisir 3 notes (sur 20). Le programme doit ensuite enregistrer chaque note sur une nouvelle ligne dans un fichier nommé `notes.txt` .

Exemple de solution :

```c
#include <stdio.h>

int main() {
    FILE *f = fopen("notes.txt", "w"); // "w" pour écraser les anciennes notes
    if (f == NULL) {
        printf("Erreur de création du fichier.\n");
        return 1;
    }

    int note;
    for (int i = 0; i < 3; i++) {
        printf("Entrez la note n°%d : ", i + 1);
        scanf("%d", &note);
        fprintf(f, "%d\n", note);
    }

    fclose(f);
    printf("Les notes ont été sauvegardées dans notes.txt\n");

    return 0;
}
```

Exercice 2 : Calculateur de moyenne

Écrivez un programme qui lit le fichier `notes.txt` créé dans l'exercice précédent. Le programme doit calculer la somme et la moyenne des notes lues, puis afficher ces résultats à l'écran.
Indice : Vous aurez besoin de `fgets` pour lire les lignes et de `atoi` (de `<stdlib.h>`) ou `sscanf` pour convertir la chaîne lue en entier.

Exemple de solution :

```c
#include <stdio.h>
#include <stdlib.h> // Pour atoi

int main() {
    FILE *f = fopen("notes.txt", "r");
    if (f == NULL) {
        printf("Impossible de lire le fichier notes.txt. Avez-vous lancé l'exercice 1 ?\
        return 1;
    }

    char buffer;
    int somme = 0;
    int nombreDeNotes = 0;

    while (fgets(buffer, sizeof(buffer), f) != NULL) {
        somme += atoi(buffer); // atoi convertit la chaîne en entier
        nombreDeNotes++;
    }

    fclose(f);

    if (nombreDeNotes > 0) {
        float moyenne = (float)somme / nombreDeNotes;
        printf("Lecture de %d notes.\n", nombreDeNotes);
        printf("Somme totale : %d\n", somme);
        printf("Moyenne : %.2f\n", moyenne);
    } else {
        printf("Le fichier ne contient aucune note.\n");
    }

    return 0;
}
```

Exercice 3 : Copie de fichier

Écrivez un programme qui copie le contenu d'un fichier source (`source.txt`) dans un fichier destination (`copie.txt`). Le programme doit lire le fichier source caractère par caractère (`fgetc`) et écrire chaque caractère dans le fichier de destination (`fputc`).
Créez manuellement un fichier `source.txt` avec quelques lignes de texte pour tester.

Exemple de solution :

```c
#include <stdio.h>

int main() {
    FILE *source = fopen("source.txt", "r");
    if (source == NULL) {
        printf("Le fichier source.txt n'a pas été trouvé.\n");
        return 1;
    }

    FILE *destination = fopen("copie.txt", "w");
    if (destination == NULL) {
        printf("Impossible de créer le fichier de destination.\n");
        fclose(source); // On n'oublie pas de fermer le premier fichier
        return 1;
    }

    int caractere;
    // fgetc retourne EOF (End Of File) à la fin du fichier
    while ((caractere = fgetc(source)) != EOF) {
        fputc(caractere, destination);
    }

    printf("La copie a été effectuée avec succès.\n");

    fclose(source);
    fclose(destination);

    return 0;
}
```

Structures et Unions

Jusqu'à présent, nous avons travaillé avec des types de données de base comme `int`, `float`, `char`, ou des collections de types identiques avec les tableaux. Mais comment représenter un objet plus complexe du monde réel, comme un étudiant, qui est défini par un nom (chaîne), un âge (entier) et une note (flottant) ?

Les **structures** sont la réponse. Elles permettent de regrouper plusieurs variables, potentiellement de types différents, en une seule unité logique.

Les Structures (`struct`)

Une structure est un type de données défini par l'utilisateur.

1. Définir une structure

On utilise le mot-clé `struct` pour définir un nouveau type. La définition crée un "modèle" ou un "plan" pour nos futures variables.

```
struct Etudiant {
    char nom;
    int age;
    float moyenne;
};
```

Cette définition ne crée aucune variable et ne réserve pas de mémoire. Elle dit simplement au compilateur : "Il existe maintenant un type appelé `struct Etudiant` , et il est composé d'un tableau de char, d'un int et d'un float."

2. Déclarer une variable de type structure

Une fois le modèle défini, on peut créer des "instances" de cette structure.

```
// Déclare une variable 'e1' de type 'struct Etudiant'
struct Etudiant e1;
```

3. Accéder aux membres

On accède aux champs (ou membres) d'une variable de structure avec l'opérateur "point" (`.`).

```c
#include <stdio.h>
#include <string.h>

// Définition de la structure
struct Etudiant {
    char nom;
    int age;
    float moyenne;
};

int main() {
    // Déclaration
    struct Etudiant e1;

    // Assignation des valeurs aux membres
    strcpy(e1.nom, "Alice");
    e1.age = 20;
    e1.moyenne = 15.5;

    // Lecture des membres
    printf("Nom de l'étudiant : %s\n", e1.nom);
    printf("Âge : %d\n", e1.age);
    printf("Moyenne : %.2f\n", e1.moyenne);

    return 0;
}
```

4. Initialisation

On peut initialiser une structure lors de sa déclaration, de la même manière qu'un tableau.

```c
struct Etudiant e2 = {"Bob", 22, 12.0};
```

Le mot-clé `typedef`

Écrire `struct Etudiant` à chaque fois peut être fastidieux. Le mot-clé `typedef` permet de créer un alias (un synonyme) pour un type. C'est une pratique extrêmement courante avec les structures.

```c
// On définit la structure ET on crée un alias 'Etudiant' en même temps
typedef struct {
    char nom;
    int age;
    float moyenne;
} Etudiant;

// Maintenant, on peut déclarer des variables beaucoup plus simplement
Etudiant e3;
Etudiant e4 = {"Charlie", 21, 18.25};
```

Pointeurs et Structures

Comme pour n'importe quelle variable, on peut avoir des pointeurs vers des structures. C'est même la manière la plus efficace de passer des structures à des fonctions pour éviter de copier de grandes quantités de données.

Pour accéder aux membres d'une structure via un pointeur, on n'utilise pas le `.` mais l'opérateur "flèche" `->`.

```c
Etudiant e = {"David", 23, 14.0};
Etudiant *ptr_e = &e; // Pointeur vers notre structure

// Accès via le pointeur
printf("Nom de l'étudiant : %s\n", ptr_e->nom);
printf("Âge : %d\n", ptr_e->age);

// ptr_e->nom est un raccourci pour (*ptr_e).nom
```

Les Unions (`union`)

Une union est syntaxiquement similaire à une structure, mais son fonctionnement est radicalement différent. Dans une union, **tous les membres partagent la même zone mémoire**.

Cela signifie qu'une variable de type union ne peut stocker qu'**une seule de ses valeurs membres à la fois**. La taille de l'union est déterminée par la taille de son plus grand membre.

```c
typedef union {
    int i;
    float f;
    char c;
} Data;

int main() {
    Data d;
    d.i = 97;
    printf("d.i = %d\n", d.i);
    printf("d.c = %c\n", d.c); // Affiche 'a', car le code ASCII de 97 est 'a'

    d.f = 3.14;
    printf("d.f = %.2f\n", d.f);
    // La valeur de d.i est maintenant corrompue car la mémoire a été réécrite !
    printf("d.i après modification de f = %d\n", d.i);
}
```

Les unions sont utiles dans des situations avancées où l'on veut économiser de la mémoire et où l'on sait qu'une seule des valeurs sera pertinente à un instant T.

Exercices

Exercice 1 : Point géométrique

1. Créez une structure `Point` (avec `typedef`) pour représenter un point dans un plan 2D, avec deux membres `int x` et `int y`.
2. Dans `main`, déclarez et initialisez une variable de type `Point`.
3. Affichez ses coordonnées sous la forme `(x, y)`.

Exemple de solution :

```c
#include <stdio.h>

typedef struct {
    int x;
    int y;
} Point;

int main() {
    Point p1 = {10, 20};

    printf("Les coordonnées du point sont (%d, %d).\n", p1.x, p1.y);

    return 0;
}
```

Exercice 2 : Fonction d'affichage

1. Créez une structure `Livre` avec un titre, un auteur (chaînes de caractères) et une année de publication (entier). Utilisez `typedef` .
2. Écrivez une fonction `afficherLivre` qui reçoit un **pointeur** vers une variable `Livre` et qui affiche ses informations de manière formatée.
3. Dans `main` , créez une instance de `Livre` , puis appelez la fonction pour l'afficher.

Exemple de solution :

```c
#include <stdio.h>

typedef struct {
    char titre;
    char auteur;
    int annee;
} Livre;

// La fonction prend un pointeur constant car elle ne modifie pas la structure
void afficherLivre(const Livre *l) {
    printf("--- Fiche du Livre ---\n");
    printf("Titre  : %s\n", l->titre);
    printf("Auteur : %s\n", l->auteur);
    printf("Année  : %d\n", l->annee);
    printf("----------------------\n");
}

int main() {
    Livre monLivre = {"Le Seigneur des Anneaux", "J.R.R. Tolkien", 1954};

    afficherLivre(&monLivre);

    return 0;
}
```

Exercice 3 : Tableau de structures

1. Reprenez la structure `Livre` de l'exercice précédent.
2. Dans `main`, déclarez un tableau capable de contenir 3 `Livre`.
3. Initialisez ce tableau avec les données de trois livres de votre choix.
4. Écrivez une boucle `for` qui parcourt le tableau et utilise la fonction `afficherLivre` pour afficher chaque livre de votre petite bibliothèque.

Exemple de solution :

```c
#include <stdio.h>

typedef struct {
    char titre;
    char auteur;
    int annee;
} Livre;

void afficherLivre(const Livre *l) {
    printf("Titre: %s, Auteur: %s, Année: %d\n", l->titre, l->auteur, l->annee);
}

int main() {
    Livre bibliotheque = {
        {"1984", "George Orwell", 1949},
        {"Dune", "Frank Herbert", 1965},
        {"Fondation", "Isaac Asimov", 1951}
    };

    printf("Contenu de ma bibliothèque :\n");
    for (int i = 0; i < 3; i++) {
        afficherLivre(&bibliotheque[i]);
    }

    return 0;
}
```

Allocation Mémoire Dynamique

Jusqu'à présent, la taille de toutes nos variables et de tous nos tableaux était fixée à la compilation. Par exemple, `int tableau[10];` réserve de la place pour 10 entiers, ni plus, ni moins. Mais que faire si nous ne connaissons pas la quantité de données à stocker à l'avance ? Par exemple, si nous voulons stocker un nombre de notes saisi par l'utilisateur ?

C'est là qu'intervient l'**allocation mémoire dynamique**. Elle permet à notre programme de demander de la mémoire au système d'exploitation **pendant son exécution**.

La Pile (Stack) vs. le Tas (Heap)

Pour comprendre l'allocation dynamique, il faut distinguer deux zones principales de la mémoire :

1. **La Pile (The Stack) :** C'est une zone de mémoire très rapide, gérée automatiquement. Toutes les variables locales que nous avons déclarées jusqu'ici (`int a;` , `char tableau[50];` à l'intérieur d'une fonction) sont stockées sur la pile. La mémoire est allouée quand on entre dans une fonction et automatiquement libérée quand on en sort. C'est simple et efficace, mais sa taille est limitée et fixée au lancement du programme.
2. **Le Tas (The Heap) :** C'est une grande zone de mémoire, beaucoup plus vaste que la pile, mais moins organisée. C'est dans cette zone que nous pouvons demander des blocs de mémoire de la taille que l'on souhaite pendant l'exécution. L'inconvénient ? La gestion de cette mémoire est **manuelle**. C'est à nous, programmeurs, de demander la mémoire et, surtout, de la **libérer** quand nous n'en avons plus besoin.

Les Fonctions Clés de `<stdlib.h>`

Pour gérer la mémoire du tas, on utilise principalement quatre fonctions qui se trouvent dans la bibliothèque `<stdlib.h>` .

1. `malloc` (Memory Allocation)

C'est la fonction de base pour allouer de la mémoire.

```
void* malloc(size_t taille);
```

- Elle prend en argument la **taille en octets** du bloc de mémoire que l'on souhaite.
- Elle retourne un **pointeur générique** `void*` vers le début de ce bloc de mémoire.
- Si le système ne peut pas allouer la mémoire demandée, elle retourne `NULL` . **Il est impératif de toujours vérifier cette valeur.**
- La mémoire allouée par `malloc` n'est **pas initialisée** ; elle contient des valeurs "poubelle".

Pour déterminer la taille nécessaire, on utilise l'opérateur `sizeof` . Par exemple, pour allouer de la place pour 5 entiers, on demandera `5 * sizeof(int)` octets.

```c
#include <stdio.h>
#include <stdlib.h> // Indispensable !

int main() {
    int *tableau = NULL;
    int taille = 5;

    // On alloue de la mémoire pour 5 entiers
    tableau = (int*) malloc(taille * sizeof(int));

    // On vérifie si l'allocation a réussi
    if (tableau == NULL) {
        printf("Erreur d'allocation mémoire.\n");
        return 1;
    }

    printf("Mémoire allouée avec succès.\n");

    // On utilise le tableau : on le remplit avec des valeurs
    printf("Remplissage du tableau...\n");
    for (int i = 0; i < taille; i++) {
        tableau[i] = i * 10; // ex: 0, 10, 20, 30, 40
    }

    // On affiche le contenu pour vérifier
    printf("Contenu du tableau : ");
    for (int i = 0; i < taille; i++) {
        printf("%d ", tableau[i]);
    }
    printf("\n");

    // Ne pas oublier de libérer la mémoire !
    printf("Libération de la mémoire.\n");
    free(tableau);
    tableau = NULL; // Bonne pratique pour éviter un pointeur suspendu

    return 0;
}
```

Notez le `(int*)` devant `malloc`. Le `void*` retourné par `malloc` est un pointeur générique. On doit le "caster" (le convertir) en un pointeur du type approprié (`int*` dans notre cas) pour pouvoir l'utiliser

correctement.

2. `free` (Libération)

C'est le pendant indispensable de `malloc`.

```
void free(void *pointeur);
```

- Elle prend en argument un pointeur qui a été retourné par `malloc`, `calloc` ou `realloc`.
- Elle **libère le bloc de mémoire**, le rendant de nouveau disponible pour le système.

Oublier d'appeler `free()` est une erreur grave appelée "fuite mémoire" (memory leak). Si votre programme alloue de la mémoire sans jamais la libérer, il consommera de plus en plus de RAM jusqu'à potentiellement ralentir ou faire planter le système. Pour chaque `malloc`, il doit y avoir un `free` correspondant.

3. `calloc` (Contiguous Allocation)

Similaire à `malloc`, mais avec deux petites différences.

```
void* calloc(size_t nombre_elements, size_t taille_element);
```

- Elle prend le nombre d'éléments et la taille de chaque élément comme deux arguments distincts.
- **Différence majeure :** Elle **initialise la mémoire allouée à zéro**. C'est plus sûr que `malloc` si vous voulez être certain de ne pas avoir de valeurs "poubelle".

```
// Alloue de la mémoire pour 10 float et les initialise tous à 0.0
float *valeurs = (float*) calloc(10, sizeof(float));
```

4. `realloc` (Re-allocation)

Permet de redimensionner un bloc de mémoire déjà alloué.

```
void* realloc(void *pointeur, size_t nouvelle_taille);
```

- Prend le pointeur du bloc à redimensionner et la nouvelle taille en octets.
- Si la nouvelle taille est plus grande, elle étend le bloc (en le déplaçant si nécessaire).
- Si la nouvelle taille est plus petite, elle le rétrécit.
- Retourne un pointeur vers le nouveau bloc (qui peut avoir une adresse différente de l'ancien !).

Exercices

Exercice 1 : Tableau dynamique

Écrivez un programme qui demande à l'utilisateur combien de notes il souhaite saisir. Allouez dynamiquement un tableau d'entiers de cette taille avec `malloc`. Remplissez le tableau avec les notes saisies par l'utilisateur, puis affichez toutes les notes. N'oubliez pas de libérer la mémoire à la fin.

Exemple de solution :

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int nombreDeNotes;
    printf("Combien de notes voulez-vous entrer ? ");
    scanf("%d", &nombreDeNotes);

    int *notes = (int*) malloc(nombreDeNotes * sizeof(int));
    if (notes == NULL) {
        return 1; // Erreur
    }

    for (int i = 0; i < nombreDeNotes; i++) {
        printf("Entrez la note n°%d : ", i + 1);
        scanf("%d", &notes[i]);
    }

    printf("\nVoici les notes que vous avez saisies :\n");
    for (int i = 0; i < nombreDeNotes; i++) {
        printf("%d ", notes[i]);
    }
    printf("\n");

    free(notes);

    return 0;
}
```

Exercice 2 : Structure dynamique

1. Définissez une structure `Personne` avec un nom (chaîne) et un âge (entier).
2. Dans `main`, allouez dynamiquement de la mémoire pour **une** `Personne` en utilisant `malloc`.
3. Remplissez les champs de la structure allouée.
4. Affichez les informations de la personne.
5. Libérez la mémoire.

Exemple de solution :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char nom;
    int age;
} Personne;

int main() {
    Personne *p = (Personne*) malloc(sizeof(Personne));
    if (p == NULL) {
        return 1;
    }

    strcpy(p->nom, "Gérard");
    p->age = 65;

    printf("Personne : %s, %d ans.\n", p->nom, p->age);

    free(p);

    return 0;
}
```

Exercice 3 : Agrandir un tableau

Écrivez un programme qui :

1. Alloue un tableau pour 3 entiers.
2. Demande à l'utilisateur de remplir ces 3 entiers.
3. Utilise `realloc` pour agrandir le tableau afin qu'il puisse contenir 5 entiers au total.

4. Demande à l'utilisateur de saisir les 2 entiers supplémentaires.
5. Affiche le contenu final du tableau de 5 entiers.
6. Libère la mémoire.

Exemple de solution :

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *nombres = (int*) malloc(3 * sizeof(int));
    if (nombres == NULL) return 1;

    printf("Veuillez entrer 3 nombres :\n");
    for (int i = 0; i < 3; i++) {
        scanf("%d", &nombres[i]);
    }

    // On réalloue pour avoir de la place pour 5 entiers
    int *nouveau_nombres = (int*) realloc(nombres, 5 * sizeof(int));
    if (nouveau_nombres == NULL) {
        free(nombres); // On libère l'ancien bloc si realloc échoue
        return 1;
    }
    nombres = nouveau_nombres; // On met à jour le pointeur

    printf("Veuillez entrer 2 nombres supplémentaires :\n");
    for (int i = 3; i < 5; i++) {
        scanf("%d", &nombres[i]);
    }

    printf("\nTableau final :\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", nombres[i]);
    }
    printf("\n");

    free(nombres);

    return 0;
}
```

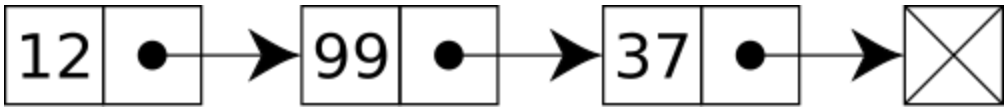
Listes Chaînées

Les tableaux, même dynamiques, ont une faiblesse : si l'on veut insérer un élément au milieu, il faut décaler tous les éléments suivants, ce qui peut être très coûteux en performance. Les **listes chaînées** sont une structure de données qui résout ce problème.

Une liste chaînée est une collection d'éléments, appelés **nœuds** (nodes), où chaque nœud contient deux choses :

1. Une **donnée** (un entier, une structure, etc.).
2. Un **pointeur** vers le nœud suivant de la liste.

Le dernier nœud de la liste a un pointeur qui pointe vers `NULL`, marquant ainsi la fin de la chaîne. L'accès à la liste se fait via un simple pointeur qui pointe vers le tout premier nœud, appelé la **tête** (head).



Le Nœud : La Brique de Base

La première étape est de définir la structure du nœud. C'est une structure auto-référentielle, car elle contient un pointeur vers son propre type.

```c
#include <stdio.h>
#include <stdlib.h>

// On utilise typedef pour plus de clarté
typedef struct Node {
    int data;           // La donnée stockée dans le nœud
    struct Node *next;  // Le pointeur vers le nœud suivant
} Node;
```

Opérations Fondamentales

1. Créer un nouveau nœud

Nous avons besoin d'une fonction qui alloue de la mémoire pour un nouveau nœud, y place une donnée, et retourne un pointeur vers ce nœud.

```
Node* createNode(int data) {
    Node *newNode = (Node*) malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Erreur d'allocation mémoire\n");
        exit(1); // Quitte le programme en cas d'échec
    }
    newNode->data = data;
    newNode->next = NULL; // Le nouveau nœud ne pointe vers rien au départ
    return newNode;
}
```

La fonction `exit()`

Dans le code ci-dessus, si `malloc` échoue, nous appelons `exit(1)`. Cette fonction, provenant de `<stdlib.h>`, termine immédiatement l'exécution de tout le programme. C'est une mesure radicale réservée aux erreurs critiques et irrécupérables, comme une allocation mémoire qui échoue. Par convention, on passe à `exit` une valeur différente de 0 (souvent 1) pour indiquer au système d'exploitation qu'une erreur s'est produite.

2. Insérer un nœud au début de la liste

C'est l'opération d'insertion la plus simple et la plus efficace.

1. Créer un nouveau nœud.
2. Faire pointer le `next` du nouveau nœud vers l'ancienne tête de la liste.
3. Mettre à jour la tête de la liste pour qu'elle pointe vers le nouveau nœud.

```c
// La fonction prend un double pointeur vers la tête
// pour pouvoir la modifier directement.
void insertAtBeginning(Node **head, int data) {
    Node *newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
}
```

3. Afficher la liste (Parcours)

Pour afficher ou parcourir la liste, on part de la tête et on suit les pointeurs `next` jusqu'à tomber sur `NULL`.

```c
void displayList(Node *head) {
    Node *current = head; // On utilise un pointeur temporaire pour parcourir
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next; // On passe au nœud suivant
    }
    printf("NULL\n");
}
```

4. Libérer la mémoire

Comme pour toute allocation dynamique, il est crucial de libérer toute la mémoire utilisée par la liste pour éviter les fuites. On doit parcourir la liste et libérer les nœuds un par un.

```c
void freeList(Node *head) {
    Node *current = head;
    Node *next;
    while (current != NULL) {
        next = current->next; // On sauvegarde le pointeur vers le suivant
        free(current);        // On libère le nœud actuel
        current = next;       // On passe au suivant
    }
}
```

Exemple complet

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node *next;
} Node;

// (Ici, on insère les définitions des fonctions createNode,
// insertAtBeginning, displayList, et freeList vues ci-dessus)

int main() {
    Node *head = NULL; // Une liste vide au départ

    insertAtBeginning(&head, 30);
    insertAtBeginning(&head, 20);
    insertAtBeginning(&head, 10);

    printf("Ma liste chaînée : ");
    displayList(head);

    freeList(head); // On nettoie la mémoire
    head = NULL;

    return 0;
}
```

Exercices

Exercice 1 : Compter les éléments

Écrivez une fonction `int countNodes(Node *head)` qui parcourt la liste et retourne le nombre de nœuds qu'elle contient.

Exemple de solution :

```c
int countNodes(Node *head) {
    int count = 0;
    Node *current = head;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}
```

Exercice 2 : Insertion à la fin

Écrivez une fonction `void insertAtEnd(Node **head, int data)` qui ajoute un nouvel élément à la **fin** de la liste.
Indice : Vous devrez parcourir la liste jusqu'au dernier élément (celui dont le `next` est `NULL`) avant d'y attacher le nouveau nœud. Pensez au cas où la liste est vide au départ.

Exemple de solution :

```c
void insertAtEnd(Node **head, int data) {
    Node *newNode = createNode(data);
    if (*head == NULL) { // Si la liste est vide
        *head = newNode;
        return;
    }
    Node *current = *head;
    while (current->next != NULL) { // On va jusqu'au dernier nœud
        current = current->next;
    }
    current->next = newNode; // On accroche le nouveau nœud
}
```

Exercice 3 : Rechercher un élément

Écrivez une fonction `Node* findNode(Node *head, int value)` qui recherche une valeur dans la liste. Si un nœud contenant cette valeur est trouvé, la fonction doit retourner un pointeur vers ce nœud. Sinon, elle doit retourner `NULL`.

Exemple de solution :

```c
Node* findNode(Node *head, int value) {
    Node *current = head;
    while (current != NULL) {
        if (current->data == value) {
            return current; // Trouvé !
        }
        current = current->next;
    }
    return NULL; // Non trouvé
}
```

Les Piles

Une **pile** (stack en anglais) est une structure de données fondamentale qui fonctionne sur le principe **LIFO** (Last-In, First-Out), c'est-à-dire "Dernier Entré, Premier Sorti".

L'analogie la plus courante est une pile d'assiettes :

- Vous ne pouvez ajouter une nouvelle assiette que sur le **dessus** de la pile.
- Vous ne pouvez retirer qu'une seule assiette à la fois : celle qui est tout **au-dessus**.

Les piles sont très utilisées en informatique, par exemple pour gérer l'historique de navigation (le bouton "page précédente" vous ramène à la dernière page visitée), ou pour le mécanisme d'appels de fonctions dans un programme.

Opérations sur une Pile

Une pile est définie par un ensemble d'opérations de base :

- **Empiler (Push)** : Ajouter un élément au sommet de la pile.
- **Dépiler (Pop)** : Retirer et retourner l'élément au sommet de la pile.
- **Sommet (Peek ou Top)** : Consulter l'élément au sommet sans le retirer.
- **Est Vide (isEmpty)** : Vérifier si la pile ne contient aucun élément.

Implémentation avec une Liste Chaînée

La liste chaînée est une excellente candidate pour implémenter une pile. L'insertion et la suppression au début d'une liste chaînée sont des opérations très rapides (en temps constant, O(1)). Le "sommet"

de notre pile sera simplement la "tête" de notre liste.

```c
#include <stdio.h>
#include <stdlib.h>

// Le nœud est identique à celui d'une liste chaînée simple
typedef struct Node {
    int data;
    struct Node *next;
} Node;

// La pile est simplement un pointeur vers le nœud du sommet
typedef struct {
    Node *top;
} Stack;

// Initialise une pile vide
Stack* createStack() {
    Stack *s = (Stack*) malloc(sizeof(Stack));
    s->top = NULL;
    return s;
}

// Vérifie si la pile est vide
int isEmpty(Stack *s) {
    return s->top == NULL;
}

// Ajoute un élément au sommet
void push(Stack *s, int data) {
    Node *newNode = (Node*) malloc(sizeof(Node));
    if (!newNode) return; // Échec de l'allocation
    newNode->data = data;
    newNode->next = s->top;
    s->top = newNode;
}

// Retire l'élément du sommet
int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Erreur : la pile est vide.\n");
        return -1; // Valeur d'erreur
    }
    Node *temp = s->top;
    int poppedData = temp->data;
```

```c
    s->top = s->top->next;
    free(temp);
    return poppedData;
}

// Affiche le contenu de la pile
void displayStack(Stack *s) {
    if (isEmpty(s)) {
        printf("Pile vide.\n");
        return;
    }
    Node *current = s->top;
    printf("Sommet -> ");
    while(current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    Stack *myStack = createStack();

    push(myStack, 10);
    push(myStack, 20);
    push(myStack, 30);

    displayStack(myStack); // Affiche : Sommet -> 30 -> 20 -> 10 -> NULL

    printf("Élément dépilé : %d\n", pop(myStack)); // Affiche 30
    printf("Élément dépilé : %d\n", pop(myStack)); // Affiche 20

    displayStack(myStack); // Affiche : Sommet -> 10 -> NULL

    // N'oubliez pas de libérer la mémoire restante !
    free(myStack->top);
    free(myStack);

    return 0;
}
```

Exercices

Exercice 1 : Inverser une chaîne

Écrivez un programme qui lit une chaîne de caractères et utilise une pile pour l'afficher à l'envers.
Indice : Empilez chaque caractère de la chaîne, puis dépilez-les un par un pour les afficher.

Exemple de solution :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// (Reprendre les définitions de Node, Stack, createStack, push, pop, isEmpty...)

int main() {
    char str[] = "Bonjour";
    Stack *charStack = createStack();

    // Empiler chaque caractère
    for (int i = 0; i < strlen(str); i++) {
        push(charStack, str[i]);
    }

    // Dépiler pour inverser
    printf("Chaîne inversée : ");
    while (!isEmpty(charStack)) {
        printf("%c", pop(charStack));
    }
    printf("\n");

    free(charStack);
    return 0;
}
```

Exercice 2 : Vérification de parenthèses

Écrivez un programme qui vérifie si une expression mathématique a des parenthèses `()` bien équilibrées.
Indice : Parcourez l'expression. Si vous voyez une `(`, empilez-la. Si vous voyez une `)`, essayez de dépiler. Si la pile est vide à ce moment, c'est une erreur. À la fin, la pile doit être vide.

Les Files d'attente

Contrairement à une pile, une **file d'attente** (queue en anglais) est une structure de données qui fonctionne sur le principe **FIFO** (First-In, First-Out), c'est-à-dire "Premier Entré, Premier Sorti".

L'analogie parfaite est une file d'attente à la caisse d'un magasin :

- Les nouvelles personnes s'ajoutent à la **fin** (ou la queue) de la file.
- La personne servie est celle qui est au **début** (ou la tête) de la file.

Les files d'attente sont très utilisées pour gérer des tâches dans l'ordre de leur arrivée, comme une file d'impression ou la gestion des requêtes sur un serveur.

Opérations sur une File

- **Enfiler (Enqueue)** : Ajouter un élément à la fin de la file.
- **Défiler (Dequeue)** : Retirer et retourner l'élément au début de la file.
- **Tête (Front/Peek)** : Consulter l'élément au début sans le retirer.
- **Est Vide (isEmpty)** : Vérifier si la file est vide.

Implémentation avec une Liste Chaînée

Pour implémenter une file efficacement, nous avons besoin d'un accès rapide à la fois au début et à la fin de la liste. Une liste chaînée est parfaite pour cela si notre structure de file contient **deux pointeurs** : un vers la tête (`front`) et un vers la queue (`rear`).

- `dequeue` se fait depuis la tête (`front`).
- `enqueue` se fait après la queue (`rear`).

Avoir ces deux pointeurs garantit que les deux opérations sont en temps constant, O(1).

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node *next;
} Node;

// La structure Queue contient un pointeur vers le début et la fin
typedef struct {
    Node *front; // Tête
    Node *rear;  // Queue
} Queue;

// Initialise une file vide
Queue* createQueue() {
    Queue *q = (Queue*) malloc(sizeof(Queue));
    q->front = q->rear = NULL;
    return q;
}

// Vérifie si la file est vide
int isEmpty(Queue *q) {
    return q->front == NULL;
}

// Ajoute un élément à la fin
void enqueue(Queue *q, int data) {
    Node *newNode = (Node*) malloc(sizeof(Node));
    if (!newNode) return;
    newNode->data = data;
    newNode->next = NULL;

    // Si la file est vide, le nouveau nœud est à la fois le début et la fin
    if (q->rear == NULL) {
        q->front = q->rear = newNode;
        return;
    }

    // Sinon, on l'ajoute après l'ancienne queue
    q->rear->next = newNode;
    q->rear = newNode; // La nouvelle queue est le nouveau nœud
}
```

```c
// Retire l'élément du début
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Erreur : la file est vide.\n");
        return -1;
    }
    Node *temp = q->front;
    int dequeuedData = temp->data;

    q->front = q->front->next;

    // Si le front devient NULL, la file est maintenant vide,
    // il faut aussi mettre à jour la queue !
    if (q->front == NULL) {
        q->rear = NULL;
    }

    free(temp);
    return dequeuedData;
}

int main() {
    Queue *myQueue = createQueue();

    enqueue(myQueue, 10);
    enqueue(myQueue, 20);
    enqueue(myQueue, 30);

    printf("Élément défilé : %d\n", dequeue(myQueue)); // Affiche 10 (le premier entré)
    printf("Élément défilé : %d\n", dequeue(myQueue)); // Affiche 20

    enqueue(myQueue, 40);

    printf("Élément défilé : %d\n", dequeue(myQueue)); // Affiche 30
    printf("Élément défilé : %d\n", dequeue(myQueue)); // Affiche 40

    // Libération de la mémoire
    free(myQueue);
    return 0;
}
```

Exercices

Exercice 1 : Simulation d'une file d'attente

Simulez une petite file d'attente. Créez une file, enfilez 5 numéros (de 1 à 5). Ensuite, défilez 2 numéros. Puis enfilez 2 autres numéros (6 et 7). Enfin, videz complètement la file en affichant chaque numéro défilé.

Exercice 2 : Compteur d'éléments

Écrivez une fonction `int count(Queue *q)` qui retourne le nombre d'éléments actuellement dans la file sans la modifier.
Indice : Il suffit de parcourir la liste chaînée du `front` jusqu'à la fin.

Types Abstraits de Données

Jusqu'ici, lorsque nous avons utilisé des structures comme les listes chaînées, nous avons manipulé directement leurs détails internes (les pointeurs `next`, les nœuds, etc.) depuis notre fonction `main`. C'est fonctionnel, mais cela crée un couplage fort : si nous décidons de changer la manière dont notre liste est implémentée, nous devons réécrire tout le code qui l'utilise.

Un **Type Abstrait de Données** (TAD), ou *Abstract Data Type* (ADT) en anglais, est une approche plus professionnelle. C'est un modèle qui définit un ensemble de données et un ensemble d'opérations sur ces données, mais qui **cache complètement les détails de l'implémentation**.

L'utilisateur du TAD sait **quoi** faire (les opérations possibles), mais pas **comment** c'est fait. Pensez à une voiture : vous savez utiliser le volant, les pédales et le levier de vitesse (l'interface), mais vous n'avez pas besoin de connaître la mécanique du moteur (l'implémentation) pour la conduire.

La Méthode en C : Fichiers d'En-tête et Fichiers Source

En C, on simule ce concept en séparant l'interface et l'implémentation dans deux fichiers :

1. **Le fichier d'en-tête (`.h`) : L'Interface Publique**
    - Déclare les fonctions que l'utilisateur a le droit d'appeler.

- Définit les types de données, souvent via un **pointeur opaque**. C'est une technique où l'on déclare une structure sans la définir, forçant l'utilisateur à ne manipuler que des pointeurs vers cette structure, sans jamais en connaître le contenu.

2. **Le fichier source (`.c`) : L'Implémentation Privée**
   - Contient la définition complète des structures de données.
   - Contient le code source des fonctions déclarées dans le fichier `.h` .

Étude de Cas : La Pile (Stack)

Une pile est un TAD très courant basé sur le principe **LIFO** (Last-In, First-Out) : le dernier élément ajouté est le premier à être retiré. Pensez à une pile d'assiettes.

1. L'interface (`pile.h`)

```c
#ifndef PILE_H
#define PILE_H

// Pointeur opaque : l'utilisateur ne saura jamais ce qu'il y a dans 'struct Pile'
typedef struct Pile Pile;

// Crée une nouvelle pile vide et retourne un pointeur vers celle-ci.
Pile* pile_creer(void);

// Détruit la pile et libère toute la mémoire.
void pile_detruire(Pile *p);

// Ajoute un élément au sommet de la pile.
void pile_pousser(Pile *p, int donnee);

// Retire et retourne l'élément au sommet de la pile.
int pile_depiler(Pile *p);

// Vérifie si la pile est vide.
int pile_est_vide(const Pile *p);

#endif // PILE_H
```

2. L'implémentation (`pile.c`)

Ici, nous choisissons d'implémenter notre pile avec une liste chaînée, mais l'utilisateur n'a pas besoin de le savoir.

```c
#include <stdio.h>
#include <stdlib.h>
#include "pile.h" // On inclut notre propre interface

// --- Définitions privées ---

typedef struct Noeud {
    int donnee;
    struct Noeud *suivant;
} Noeud;

// C'est ici qu'on définit VRAIMENT la structure Pile
struct Pile {
    Noeud *sommet; // La tête de notre liste chaînée
};

// --- Implémentation des fonctions publiques ---

Pile* pile_creer(void) {
    Pile *p = (Pile*) malloc(sizeof(Pile));
    if (p != NULL) {
        p->sommet = NULL;
    }
    return p;
}

void pile_detruire(Pile *p) {
    if (p == NULL) return;
    while (!pile_est_vide(p)) {
        pile_depiler(p);
    }
    free(p);
}

void pile_pousser(Pile *p, int donnee) {
    if (p == NULL) return;
    Noeud *nouveau = (Noeud*) malloc(sizeof(Noeud));
    if (nouveau == NULL) return;

    nouveau->donnee = donnee;
    nouveau->suivant = p->sommet;
    p->sommet = nouveau;
}
```

```c
int pile_depiler(Pile *p) {
    if (pile_est_vide(p)) {
        fprintf(stderr, "Erreur : la pile est vide.\n");
        exit(EXIT_FAILURE);
    }

    Noeud *ancien_sommet = p->sommet;
    int donnee = ancien_sommet->donnee;

    p->sommet = ancien_sommet->suivant;
    free(ancien_sommet);

    return donnee;
}

int pile_est_vide(const Pile *p) {
    return (p == NULL || p->sommet == NULL);
}
```

3. Utilisation (`main.c`)

Le code de l'utilisateur est maintenant beaucoup plus propre et ne dépend que de l'interface.

```c
#include <stdio.h>
#include "pile.h" // On inclut seulement l'interface

int main() {
    Pile *ma_pile = pile_creer();

    printf("On empile 10, 20, 30...\n");
    pile_pousser(ma_pile, 10);
    pile_pousser(ma_pile, 20);
    pile_pousser(ma_pile, 30);

    printf("On dépile : %d\n", pile_depiler(ma_pile)); // 30
    printf("On dépile : %d\n", pile_depiler(ma_pile)); // 20

    pile_detruire(ma_pile);

    return 0;
}
```

Pour compiler ce projet, on compilerait `pile.c` et `main.c` ensemble :

```
gcc main.c pile.c -o programme
```

Exercices

Exercice 1 : Taille de la pile

Ajoutez une fonction `int pile_taille(const Pile *p);` à votre TAD Pile. Elle doit retourner le nombre d'éléments actuellement dans la pile. Vous devrez modifier le `.h` et le `.c` .
Indice : Vous pouvez soit parcourir la liste à chaque appel, soit ajouter un champ `taille` dans la `struct Pile` et le mettre à jour à chaque `pousser` / `depiler` .

Exemple de solution (avec un champ `taille`) :

```c
#include <stdio.h>
#include <stdlib.h>

// --- Interface (simulée, normalement dans pile.h) ---
typedef struct Pile Pile;
Pile* pile_creer(void);
void pile_detruire(Pile *p);
void pile_pousser(Pile *p, int donnee);
int pile_depiler(Pile *p);
int pile_est_vide(const Pile *p);
int pile_taille(const Pile *p); // Nouvelle fonction

// --- Implémentation (simulée, normalement dans pile.c) ---

typedef struct Noeud {
    int donnee;
    struct Noeud *suivant;
} Noeud;

// La struct Pile est maintenant modifiée pour inclure la taille
struct Pile {
    Noeud *sommet;
    int taille;
};

Pile* pile_creer(void) {
    Pile *p = (Pile*) malloc(sizeof(Pile));
    if (p != NULL) {
        p->sommet = NULL;
        p->taille = 0; // Initialisation de la taille
    }
    return p;
}

int pile_est_vide(const Pile *p) {
    return (p == NULL || p->sommet == NULL);
}

void pile_pousser(Pile *p, int donnee) {
    if (p == NULL) return;
    Noeud *nouveau = (Noeud*) malloc(sizeof(Noeud));
    if (nouveau == NULL) return;
```

```c
    nouveau->donnee = donnee;
    nouveau->suivant = p->sommet;
    p->sommet = nouveau;
    p->taille++; // Mise à jour de la taille
}

int pile_depiler(Pile *p) {
    if (pile_est_vide(p)) {
        fprintf(stderr, "Erreur : la pile est vide.\n");
        exit(EXIT_FAILURE);
    }

    Noeud *ancien_sommet = p->sommet;
    int donnee = ancien_sommet->donnee;

    p->sommet = ancien_sommet->suivant;
    free(ancien_sommet);
    p->taille--; // Mise à jour de la taille

    return donnee;
}

// Implémentation de la nouvelle fonction
int pile_taille(const Pile *p) {
    if (p == NULL) {
        return 0;
    }
    return p->taille;
}

void pile_detruire(Pile *p) {
    if (p == NULL) return;
    while (!pile_est_vide(p)) {
        pile_depiler(p);
    }
    free(p);
}

// --- Utilisation (main.c) ---
int main() {
    Pile *ma_pile = pile_creer();
    printf("Taille initiale de la pile : %d\n", pile_taille(ma_pile));
```

```c
    printf("On empile 10, 20, 30...\n");
    pile_pousser(ma_pile, 10);
    pile_pousser(ma_pile, 20);
    pile_pousser(ma_pile, 30);
    printf("Nouvelle taille de la pile : %d\n", pile_taille(ma_pile));

    printf("On dépile : %d\n", pile_depiler(ma_pile));
    printf("Taille après un dépilage : %d\n", pile_taille(ma_pile));

    pile_detruire(ma_pile);
    return 0;
}
```

Exercice 2 : Le TAD File (Queue)

Sur le modèle de la Pile, concevez l'interface (`file.h`) et l'implémentation (`file.c`) d'un TAD **File** (Queue), qui fonctionne sur le principe **FIFO** (First-In, First-Out).
Les opérations devraient être :

- `file_creer()`
- `file_detruire()`
- `file_enfiler(File *f, int donnee)` (ajoute à la fin)
- `file_defiler(File *f)` (retire au début)
- `file_est_vide()`

*Indice pour l'implémentation : une liste chaînée est efficace si vous gardez un pointeur vers la **tête** (pour défiler) et un pointeur vers la **queue** (pour enfiler).

Exemple de solution :

```c
#include <stdio.h>
#include <stdlib.h>

// --- Définition du TAD File (Queue) ---

typedef struct Noeud {
    int donnee;
    struct Noeud *suivant;
} Noeud;

typedef struct {
    Noeud *tete;  // Début de la file (pour défiler)
    Noeud *queue; // Fin de la file (pour enfiler)
} File;

// Crée une file vide
File* file_creer(void) {
    File *f = (File*) malloc(sizeof(File));
    if (f != NULL) {
        f->tete = NULL;
        f->queue = NULL;
    }
    return f;
}

// Vérifie si la file est vide
int file_est_vide(const File *f) {
    return (f == NULL || f->tete == NULL);
}

// Ajoute un élément à la fin de la file (enfiler)
void file_enfiler(File *f, int donnee) {
    if (f == NULL) return;
    Noeud *nouveau = (Noeud*) malloc(sizeof(Noeud));
    if (nouveau == NULL) {
        perror("Allocation du nœud impossible");
        return;
    }
    nouveau->donnee = donnee;
    nouveau->suivant = NULL;

    if (file_est_vide(f)) {
        // Si la file est vide, le nouveau nœud est à la fois la tête et la queue
```

```
        f->tete = nouveau;
        f->queue = nouveau;
    } else {
        // Sinon, on l'ajoute après l'ancienne queue
        f->queue->suivant = nouveau;
        f->queue = nouveau; // La nouvelle queue est le nouveau nœud
    }
}

// Retire et retourne l'élément au début de la file (défiler)
int file_defiler(File *f) {
    if (file_est_vide(f)) {
        fprintf(stderr, "Erreur : la file est vide.\n");
        exit(EXIT_FAILURE);
    }

    Noeud *ancienne_tete = f->tete;
    int donnee = ancienne_tete->donnee;

    f->tete = ancienne_tete->suivant;
    // Si la tête devient NULL, la file est vide, donc la queue aussi
    if (f->tete == NULL) {
        f->queue = NULL;
    }

    free(ancienne_tete);
    return donnee;
}

// Libère toute la mémoire utilisée par la file
void file_detruire(File *f) {
    if (f == NULL) return;
    while (!file_est_vide(f)) {
        file_defiler(f);
    }
    free(f);
}

// --- Utilisation du TAD File ---

int main() {
    File *ma_file = file_creer();
```

```
    printf("On enfile 10, 20, puis 30 (FIFO).\n");
    file_enfiler(ma_file, 10);
    file_enfiler(ma_file, 20);
    file_enfiler(ma_file, 30);

    printf("Élément défilé : %d\n", file_defiler(ma_file)); // Doit être 10
    printf("Élément défilé : %d\n", file_defiler(ma_file)); // Doit être 20

    printf("\nOn enfile 40 et 50.\n");
    file_enfiler(ma_file, 40);
    file_enfiler(ma_file, 50);

    printf("\nOn vide la file :\n");
    while (!file_est_vide(ma_file)) {
        printf("Élément défilé : %d\n", file_defiler(ma_file));
    }

    file_detruire(ma_file);
    printf("\nLa file a été détruite.\n");

    return 0;
}
```

Exercice 3 : Changer l'implémentation

Imaginez que votre Pile est utilisée dans des millions de lignes de code. Maintenant, vous réalisez qu'une implémentation avec un tableau dynamique (`realloc`) serait plus performante pour votre cas d'usage. Comment feriez-vous la modification ?
Réponse (conceptuelle) : Vous ne toucheriez **jamais** à `pile.h` ! Vous réécririez entièrement `pile.c` en utilisant un tableau et un indice de sommet. Tout le code utilisateur continuerait de fonctionner sans aucune modification, car il ne dépend que de l'interface, qui n'a pas changé. C'est toute la puissance des TAD.

La Récursivité

La récursivité est une approche de résolution de problèmes où la solution dépend de solutions à des instances plus petites du même problème. En programmation, cela se traduit par une **fonction qui s'appelle elle-même**.

C'est une alternative aux boucles (`for` , `while`) pour effectuer des tâches répétitives. Si une boucle est une répétition "horizontale", la récursivité est une répétition "verticale", en profondeur.

Les Deux Piliers de la Récursivité

Toute fonction récursive correcte doit impérativement comporter deux éléments :

1. **Le Cas de Base (Base Case) :** C'est la condition d'arrêt. Un problème si simple qu'il peut être résolu directement, sans avoir besoin de s'appeler à nouveau. Sans cas de base, la fonction s'appellerait à l'infini, menant à une erreur de "dépassement de pile" (stack overflow).
2. **L'Étape Récursive (Recursive Step) :** C'est la partie où la fonction se rappelle elle-même, mais en lui passant une version "plus petite" ou "plus simple" du problème. L'objectif est de se rapprocher progressivement du cas de base.

Exemple Classique : La Factorielle

La factorielle d'un nombre entier `n` , notée `n!` , est le produit de tous les entiers de 1 à `n` .

- `5! = 5 * 4 * 3 * 2 * 1 = 120`

On peut définir la factorielle de manière récursive :

- `n! = n * (n-1)!` **(Étape récursive)**
- `0! = 1` **(Cas de base)**

Cette définition se traduit presque littéralement en une fonction C :

```c
#include <stdio.h>

int factorielle(int n) {
    // Cas de base : si n est 0, la factorielle est 1.
    if (n == 0) {
        return 1;
    }
    // Étape récursive : n * la factorielle du nombre juste avant.
    else {
        return n * factorielle(n - 1);
    }
}

int main() {
    int nombre = 5;
    printf("La factorielle de %d est %d.\n", nombre, factorielle(nombre));
    return 0;
}
```

Comment ça marche pour `factorielle(3)` ?

1. `factorielle(3)` est appelée. `3 != 0`, donc elle retourne `3 * factorielle(2)`.
2. `factorielle(2)` est appelée. `2 != 0`, donc elle retourne `2 * factorielle(1)`.
3. `factorielle(1)` est appelée. `1 != 0`, donc elle retourne `1 * factorielle(0)`.
4. `factorielle(0)` est appelée. `0 == 0`, elle atteint le cas de base et retourne `1`.
5. Le résultat remonte : `factorielle(1)` devient `1 * 1 = 1`.
6. Le résultat remonte : `factorielle(2)` devient `2 * 1 = 2`.
7. Le résultat remonte : `factorielle(3)` devient `3 * 2 = 6`.

Récursivité vs. Itération (Boucles)

Chaque problème récursif peut être résolu avec une boucle, et vice-versa.

Factorielle avec une boucle :

```c
int factorielle_iterative(int n) {
    int resultat = 1;
    for (int i = 1; i <= n; i++) {
        resultat = resultat * i;
    }
    return resultat;
}
```

Alors, quand choisir la récursivité ?

- **Avantages de la récursivité :**
  - **Élégance et lisibilité :** Pour les problèmes qui sont naturellement récursifs (parcours d'arbres, algorithmes "diviser pour régner"), le code récursif est souvent beaucoup plus court, propre et facile à comprendre.
  - **Simplicité :** Évite la gestion manuelle de structures de données complexes (comme une pile) pour suivre l'état.
- **Inconvénients de la récursivité :**
  - **Performance :** Chaque appel de fonction a un coût (création d'un contexte sur la pile). Les boucles sont généralement plus rapides.
  - **Consommation mémoire :** Chaque appel récursif consomme de la mémoire sur la pile. Une récursion trop profonde peut saturer la pile et faire planter le programme (stack overflow).

Exercices

Exercice 1 : Somme des entiers

Écrivez une fonction récursive `somme(n)` qui calcule la somme de tous les entiers de 1 à `n`.
La définition récursive est : `somme(n) = n + somme(n-1)`, avec le cas de base `somme(1) = 1`.

Exemple de solution :

```c
#include <stdio.h>

int somme(int n) {
    // Cas de base
    if (n <= 1) {
        return 1;
    }
    // Étape récursive
    else {
        return n + somme(n - 1);
    }
}

int main() {
    printf("La somme des entiers de 1 à 10 est %d.\n", somme(10)); // 55
    return 0;
}
```

Exercice 2 : Puissance d'un nombre

Écrivez une fonction récursive `puissance(base, exp)` qui calcule `base` élevée à la puissance `exp`. Définition : `puissance(b, e) = b * puissance(b, e-1)`, cas de base `puissance(b, 0) = 1`.

Exemple de solution :

```c
#include <stdio.h>

long puissance(int base, int exp) {
    // Cas de base
    if (exp == 0) {
        return 1;
    }
    // Étape récursive
    else {
        return base * puissance(base, exp - 1);
    }
}

int main() {
    printf("2^10 = %ld\n", puissance(2, 10)); // 1024
    return 0;
}
```

Exercice 3 : La suite de Fibonacci

La suite de Fibonacci est définie par :

- `fib(0) = 0`
- `fib(1) = 1`
- `fib(n) = fib(n-1) + fib(n-2)` pour `n > 1`

Écrivez une fonction récursive `fibonacci(n)` qui calcule le n-ième terme de la suite.

Exemple de solution :

```c
#include <stdio.h>

int fibonacci(int n) {
    // Deux cas de base
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    // Étape récursive avec deux appels
    else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int main() {
    // La suite commence : 0, 1, 1, 2, 3, 5, 8, 13...
    printf("Le 8ème terme de Fibonacci est %d.\n", fibonacci(8)); // 21
    return 0;
}
```

(Note : cette implémentation de Fibonacci est très élégante mais très inefficace car elle recalcule de nombreuses fois les mêmes valeurs. C'est un exemple classique des compromis de la récursivité.)

Analyse d algorithmes

Écrire un programme qui fonctionne est une chose. Écrire un programme qui fonctionne **efficacement** en est une autre. L'analyse d'algorithmes est l'étude de la performance des algorithmes : combien de temps prennent-ils pour s'exécuter et combien de mémoire consomment-ils ?

Cette analyse nous permet de comparer objectivement différentes solutions à un même problème et de prédire comment un algorithme se comportera lorsque la taille des données en entrée augmente.

La Complexité Temporelle et la Notation Big O

Le temps d'exécution d'un algorithme dépend de nombreux facteurs (vitesse du processeur, langage, compilateur...). Pour s'abstraire de ces détails, on ne mesure pas le temps en secondes, mais en

nombre d'opérations élémentaires (affectations, comparaisons, calculs...) que l'algorithme effectue en fonction de la taille de son entrée, notée `n`.

La **notation Big O** (Grand O) est utilisée pour décrire le **comportement asymptotique** d'une fonction, c'est-à-dire comment elle se comporte lorsque `n` devient très grand. Elle nous donne un ordre de grandeur de la croissance du temps d'exécution.

Complexités Courantes (de la plus rapide à la plus lente)

1. O(1) - Temps Constant

L'algorithme prend le même temps, quelle que soit la taille de l'entrée `n`.

- **Exemple :** Accéder à un élément d'un tableau par son indice (`tableau[i]`).

```
int getFirstElement(int arr[], int n) {
    return arr; // Toujours 1 opération, que n soit 10 ou 1,000,000
}
```

2. O(log n) - Temps Logarithmique

Le temps d'exécution augmente très lentement. À chaque étape, l'algorithme divise la taille du problème par une constante.

- **Exemple :** La recherche dichotomique (binary search) dans un tableau trié.

3. O(n) - Temps Linéaire

Le temps d'exécution est directement proportionnel à la taille de l'entrée. Si `n` double, le temps double.

- **Exemple :** Parcourir tous les éléments d'un tableau.

```c
int findMax(int arr[], int n) {
    int max = arr;
    for (int i = 1; i < n; i++) { // La boucle s'exécute n-1 fois
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

4. O(n log n) - Temps "Linéarithmique"

Très courant pour les algorithmes de tri efficaces. C'est un compromis excellent.

- **Exemples :** Le tri fusion (Merge Sort), le tri rapide (Quick Sort).

5. O(n²) - Temps Quadratique

Le temps d'exécution est proportionnel au carré de la taille de l'entrée. Si `n` double, le temps est multiplié par 4. Ces algorithmes deviennent très lents pour de grandes valeurs de `n`.

- **Exemple :** Une boucle imbriquée qui parcourt toutes les paires d'éléments d'un tableau.

```c
void printAllPairs(int arr[], int n) {
    for (int i = 0; i < n; i++) {       // Boucle externe : n tours
        for (int j = 0; j < n; j++) {   // Boucle interne : n tours
            printf("(%d, %d)\n", arr[i], arr[j]); // Exécuté n*n fois
        }
    }
}
```

6. O(2ⁿ) - Temps Exponentiel

L'algorithme devient extrêmement lent très rapidement. Souvent associé à la récursivité "naïve" qui explore toutes les possibilités.

- **Exemple :** Le calcul de Fibonacci avec la récursivité simple (vue au chapitre précédent).

La Complexité Spatiale

La complexité spatiale mesure la quantité de **mémoire supplémentaire** (en plus des données d'entrée) qu'un algorithme utilise en fonction de la taille de l'entrée `n`. La notation Big O s'applique de la même manière.

- Un algorithme qui n'utilise que quelques variables simples a une complexité spatiale **O(1)**.
- Un algorithme qui crée une copie du tableau d'entrée a une complexité spatiale **O(n)**.
- Une fonction récursive a une complexité spatiale liée à sa profondeur maximale sur la pile d'appels.

Le Compromis Espace-Temps

En informatique, il est rare d'avoir le meilleur des deux mondes. On fait souvent face à un **compromis espace-temps** (space-time tradeoff) :

- Un algorithme peut être très rapide mais consommer beaucoup de mémoire.
- Un autre peut être très économe en mémoire mais beaucoup plus lent.

Le choix de l'algorithme dépendra des contraintes du problème : la vitesse est-elle plus critique que la consommation de RAM ?

Exercices

Exercice 1 : Quelle complexité ?

Analysez la complexité temporelle (en notation Big O) des fonctions suivantes.

a)

```c
void checkValue(int arr[], int n, int value) {
    if (arr == value) {
        printf("Trouvé !\n");
    } else if (arr[n-1] == value) {
        printf("Trouvé !\n");
    }
}
```

b)

```
int sumMatrix(int matrix) {
    int sum = 0;
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            sum += matrix[i][j];
        }
    }
    return sum;
}
```

c)

```
int countOccurrences(int arr[], int n, int value) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] == value) {
            count++;
        }
    }
    return count;
}
```

Solutions :

a) O(1) : Le nombre d'opérations (deux comparaisons) est constant et ne dépend pas de la taille `n` du tableau.

b) O(1) : La taille de la matrice est fixée à 10x10. Les boucles s'exécutent toujours 100 fois, c'est une constante. La complexité ne dépend pas d'une variable `n` .

c) O(n) : La boucle parcourt tout le tableau une fois. Le nombre d'opérations est linéaire par rapport à `n` .

Exercice 2 : Comparaison de recherches

Vous disposez d'un tableau d'un million d'entiers.

1. Si le tableau n'est pas trié, quelle est la complexité (dans le pire des cas) pour trouver si un nombre `x` est présent ?
2. Si le tableau est trié, quelle est la complexité (dans le pire des cas) pour la même recherche ?
3. Lequel est le plus rapide pour de grandes valeurs de `n` ?

Solutions :

1. **O(n)** : Il faut potentiellement vérifier chaque élément un par un (recherche linéaire).
2. **O(log n)** : On peut utiliser la recherche dichotomique.
3. **O(log n)** est beaucoup plus rapide. Pour n = 1,000,000, `n` est 1,000,000 alors que $\log_2(n)$ est environ 20. C'est une différence énorme !

Exercice 3 : Complexité spatiale

Quelle est la complexité spatiale de la fonction suivante ?

```c
int* createCopy(int arr[], int n) {
    int *copy = (int*) malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        copy[i] = arr[i];
    }
    return copy;
}
```

Solution :

O(n) : La fonction alloue un nouveau tableau dont la taille est directement proportionnelle à la taille `n` de l'entrée.