C - #include

En langage C, un programme ne se limite pas à un simple fichier. Dès que la taille d'un projet augmente, il devient indispensable de **structurer le code**, de **réutiliser des fonctions existantes** et de **séparer les responsabilités**

5TTR 6TTR Découverte

Le langage C repose pour cela sur un système de **bibliothèques** et de **fichiers organisés** (`.h` et `.c`), reliés entre eux grâce à la directive `#include`. Comprendre ce mécanisme est fondamental : il permet non seulement d'utiliser les fonctions standard du langage, mais aussi de créer ses propres bibliothèques, claires, maintenables et professionnelles.

Ce chapitre explique comment fonctionne `#include`, à quoi servent les librairies, et comment organiser correctement un programme C en plusieurs fichiers.

À quoi servent les bibliothèques en C ?

En C, **on ne réécrit pas tout soi-même**. Le langage met à disposition des **bibliothèques** : des ensembles de fonctions déjà écrites, testées et optimisées.

Exemples de fonctions fournies par des bibliothèques :

- `printf()` → affichage à l'écran
- `scanf()` → lecture clavier
- `sqrt()` → racine carrée
- `strlen()` → longueur d'une chaîne

Ces fonctions **n'existent pas par magie** : il faut dire au compilateur **où elles sont déclarées**.

Le rôle de `#include`

La directive `#include` sert à **insérer le contenu d'un fichier dans un autre fichier** avant la compilation.

Exemple :

```
1 | #include <stdio.h>
```

Cela signifie :

> « Inclure les déclarations des fonctions d'entrée/sortie standard »

Sans cette ligne :

```c
printf("Bonjour");
```

le compilateur **ne sait pas ce qu'est** `printf`.

`#include <...>` ou `#include "..."` : quelle différence ?

3.1 `#include <stdio.h>`

- Utilisé pour les **bibliothèques standard**
- Le compilateur cherche le fichier dans ses dossiers système

Exemples courants :

```c
#include <stdio.h>   // printf, scanf
#include <stdlib.h>  // malloc, rand, exit
#include <math.h>    // sqrt, pow
#include <string.h>  // strlen, strcmp
```

3.2 `#include "monfichier.h"`

- Utilisé pour **vos propres fichiers**
- Le compilateur cherche d'abord dans le dossier du projet

Exemple :

```c
#include "utils.h"
```

Qu'est-ce qu'un fichier `.h` ?

Un fichier `.h` (header / en-tête) **déclare ce qui existe**, mais ne contient pas le code complet.

Il sert à :

- déclarer des **fonctions**
- déclarer des **constantes**
- déclarer des **types**

Exemple : `math_utils.h`

```c
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int addition(int a, int b);
int soustraction(int a, int b);

#endif
```

➡️ Ici, on **annonce** que ces fonctions existent quelque part.

Qu'est-ce qu'un fichier `.c` ?

Le fichier `.c` **contient le code réel**, c'est-à-dire les **définitions des fonctions**.

Exemple : `math_utils.c`

```c
#include "math_utils.h"

int addition(int a, int b) {
    return a + b;
}

int soustraction(int a, int b) {
    return a - b;
}
```

➡️ Le `.c` implémente ce qui a été annoncé dans le `.h`.

Pourquoi séparer `.h` et `.c` ?

Bonne pratique professionnelle

Cette séparation permet :

- une meilleure **lisibilité**
- une **réutilisation** du code
- un **travail en équipe**
- une **compilation plus claire**

Règle simple à retenir

> **DANS LE `.H` : CE QUE LE PROGRAMME PEUT UTILISER** **Dans le `.c` : comment cela fonctionne**

Exemple complet avec `main.c`

`main.c`

```c
#include <stdio.h>
#include "math_utils.h"

int main(void) {
    int resultat;

    resultat = addition(5, 3);
    printf("Addition : %d\n", resultat);

    resultat = soustraction(10, 4);
    printf("Soustraction : %d\n", resultat);

    return 0;
}
```

Le rôle du `#ifndef` / `#define` / `#endif`

Dans un `.h`, ces lignes évitent **l'inclusion multiple**.

```c
#ifndef MATH_UTILS_H
#define MATH_UTILS_H
...
#endif
```

Sans ça :

- un même fichier pourrait être inclus plusieurs fois
- cela provoquerait des **erreurs de compilation**

➡️ C'est une **protection obligatoire** dans tout fichier `.h`.

Compilation avec plusieurs fichiers

Commande classique avec GCC :

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

Explication :

- `main.c` → programme principal
- `math_utils.c` → fonctions externes
- `-o programme` → nom de l'exécutable

⚠️ **On ne compile jamais les** `.h`, seulement les `.c`.

Erreurs fréquentes à éviter

- Mettre du code de fonction dans un `.h`
- Oublier d'inclure son `.h` dans le `.c`
- Déclarer une fonction différemment dans le `.h` et le `.c`
- Inclure un `.c` dans un autre `.c`
- Oublier les gardes `#ifndef`

À retenir (synthèse)

- `#include` insère du code avant compilation
- `<...>` → bibliothèques standard
- `"..."` → fichiers du projet
- `.h` → déclarations
- `.c` → implémentations
- Chaque `.c` est compilé séparément
- Le `.h` sert de **contrat**