Structure de projet

Programmation modulaire

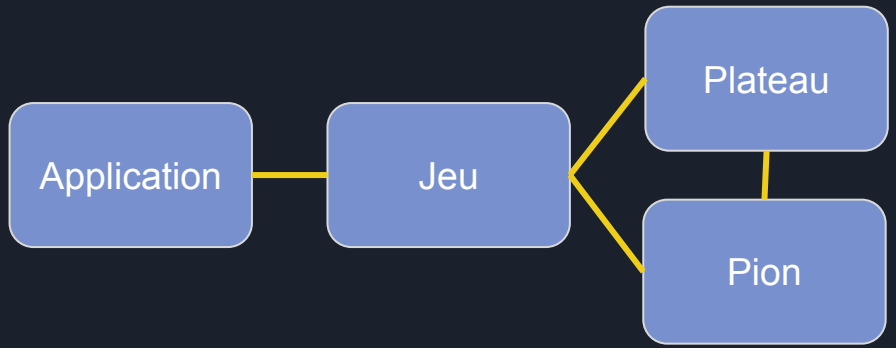
Un projet C/C++ est architecturé de manière **modulaire** (comme en C#).

Les fonctionnalités du programme sont séparées en modules codés dans des **fichiers séparés**.

Programmation modulaire

Chaque module contient un ensemble de **fonctions**, de **types** et de **variables** représentant une facette bien spécifique du programme.

Il est important de bien définir en amont les **liens** entre modules.



Module

Pour chaque module, on écrira :

- Un fichier header (.h),
- Un fichier source (.cpp).

Header

Le header contient toutes les **déclarations** nécessaires pour utiliser le module depuis l'extérieur :

- **Types** : enums, typedefs, structures, ...
- **Prototypes de fonctions** sans définition
- **Variables globales** (mot-clé extern)

Les informations internes au fonctionnement du module n'apparaissent pas dans le header.

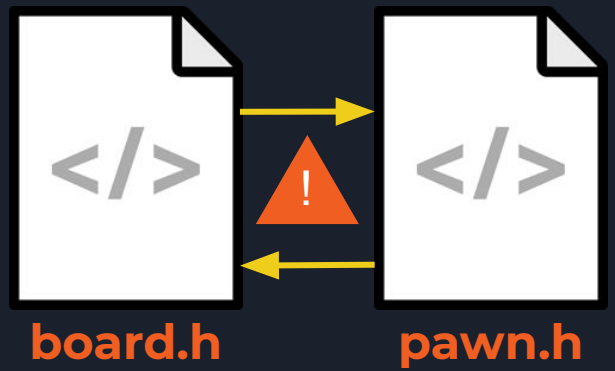
Header

Le header doit être protégé contre la **double inclusion** et les **inclusions cycliques** :

```
#pragma once
// Header code
```

Ou

```
#ifndef GAME_H
#define GAME_H
// Header code
#endif
```



Header

Le fichier header d'un module sera **inclus** dans :

- Le fichier source du module,
- Les fichiers sources des autres modules connaissant ce module,
- Aussi peu de headers que possible !

Sources

Un fichier source contient essentiellement :

- La **définition des fonctions** déclarées par le header,
- L'**initialisation des variables globales** déclarées par le header,
- Les **fonctions, types et variables internes** au fonctionnement du fichier source.

Sources

Un fichier source **inclut toujours son header associé en première inclusion** !

Il inclut ensuite les headers des modules qu'il connaît.

Un fichier source n'est jamais inclus !

Variables globales

Pour construire une architecture propre, on **évitera autant que possible** les variables globales !

On utilisera essentiellement des **variables globales constantes** (mot-clé `const`).

Données

Les **classes et structures** vont servir à architecturer les données.

On utilisera les **pointeurs et références** pour **transmettre** des données structurées d'un module à un autre.

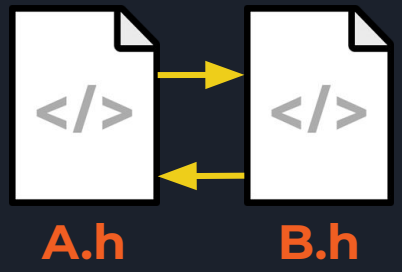
Par exemple, pour déplacer un pion sur un plateau :

```
void movePawn(Board &board, Pawn &pawn, int x, int y);
```

Références croisées

Un header A.h déclare un type A et a besoin de connaître un type B. Alors, A.h inclus B.h.

Un header B.h déclare un type B et a besoin de connaître un type A. Alors, B.h inclus A.h.



Il y a **références croisées**. Le compilateur ne peut pas résoudre cette situation (soit A, soit B n'est pas défini avant d'être utilisé).

Références croisées

```
#pragma once // 1- Compile A.h en 1er
#include "B.h" // 2- Inclut B.h ici
struct A {
    B val;
};
```

```
#pragma once
#include "A.h" // 3- Ne réinclut pas A.h
struct B {
    A val; // 4 - A n'est pas encore défini !
}; // Erreur de compilation
```

Références croisées

Dans certains cas, la référence croisée peut être résolue par un type **déclaré mais non-défini**.

Il est possible de déclarer une structure ou une classe non-définie et d'utiliser des **pointeurs vers ce type**.

On ne peut pas déclarer de variables d'un type non-défini mais on peut déclarer des pointeurs vers ce type.

Références croisées

```
#pragma once // 1- Compile A.h en 1er
#include "B.h" // 2- Inclut B.h ici
struct A {
    B val;
}; // 5- Ok
```

```
#pragma once
struct A; // 3- Déclare A
struct B {
    A *val;
};
```