7. S'initier à la programmation modulaire

Jusqu'à présent, nous avons travaillé dans un unique fichier, car nos programmes étaient tous petits ! La plupart des programmes contiennent plusieurs fonctions, et il faut les garder organisés ; nous allons donc maintenant apprendre les bases de la **programmation modulaire.**

En temps normal, les fonctions sont placées dans ou avant le `main` ; l'ordinateur connaît la fonction et sait où aller la chercher. Cependant, si on la positionne après le main, la machine sera perdue et ne pourra pas compiler le code !

La programmation modulaire nous permet de contourner ce soucis.

H1: Les Prototypes De Fonctions

Pour pouvoir appeller une fonction où qu'elle se trouve dans le programme, nous allons utiliser des **prototypes.**

Prenons la fonction `aire` du programme qui nous permettait de calculer l'aire des rectangles.

```
int aire(int largeur, int longueur)
{
    return largeur * longueur;
}
```

Pour la transformer en prototype, il suffit de prendre la première ligne et de la placer juste après les #include, assortie d'un point virgule, comme suit :

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

int aire(int largeur, int longueur); // prototype
```

📝 Le prototype est une indication pour l'ordinateur qui lui permet de s'organiser et de savoir qu'il existe une fonction qui prend les paramètres qu'on indique en entrée et renvoie une sortie du type spécifié. Toutes les fonctions ont besoin d'un prototype, à l'exception de `main` qui est connue des ordinateurs et est présente **dans tous les programmes**.

⚠ Il faut **toujours** rédiger les prototypes de nos fonctions ;

⚠ Il ne faut **JAMAIS** oublier le `;` à la fin d'une fonction ; c'est ce qui permet à l'ordinateur de la distinguer du vrai début d'une fonction. Sans le point virgule, on risque d'avoir des erreur incompréhensibles à la compilation.

H1: Structurer Un Projet

Un projet est composé par l'ensemble des fichiers sources de notre programme. Il est *très important* de rester bien organisé, pour que la machine puisse s'organiser, et pour ne pas se perdre dans son programme ; c'est

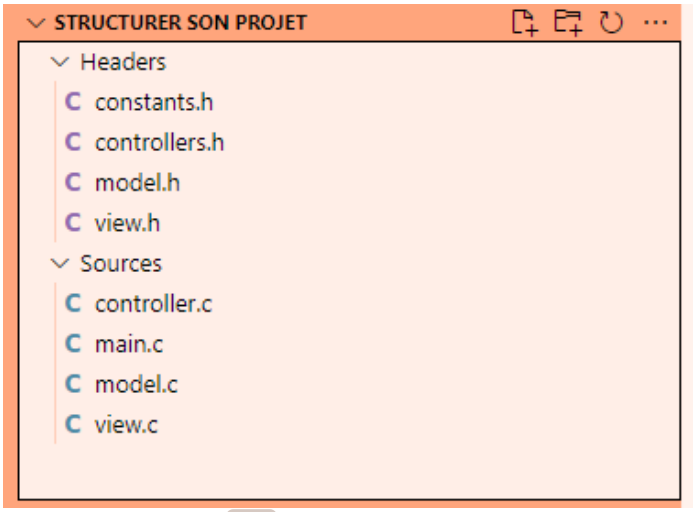
d'autant plus important si on travaille à plusieur sur un projet !

On détaille ci-après la bonne façon de structurer un projet à l'aide de *headers* (ou fichiers d'en-tête en français).

H2 : Créer plusieurs fichier par projet

⚠ Il faut toujours avec un fichier `main.c`, qui contient la fonction `main`.

Voici comment on structure un projet :



On note qu'on regroupe dans un même fichier les fonctions par thème :

- dans view.c, on groupe les fonctions relatives à l'interface graphique ;
- dans controller.c, on regroupe les fonctions relatives à la logique de notre projet...

On voit qu'on a créé deux folders : `Headers` et `Sources`, qui continennent des fichiers dont les extensions diffièrent : `.h` et `.c`. Voici en quoi ils diffèrent :

1. Les fichiers `.h` sont des fichiers header, qui contiennent les prototypes des fonctions ;
2. Les fichiers `.c` sont les fichiers sources, qu'on connaît déjà ; ils contiennent les fonctions elles-mêmes.

⚠ On évite de mettre des prototypes dans les fichiers `.c`, sauf si notre programme est riquiqui minuscule.

Notons également que chaque fichier `.c` a son équivalent `.h`, qui contient les prototypes des fonctions utilisées.

Pour pouvoir utiliser les prototypes des fichiers `.h`, il faut inclure ce dernier dans le code `.c` grâce à une directice de préprocesseur - `#include` !

Prenons par exemple notre fichier `view.c` : pour inclure les protypes de fonction liées, on écrit :

```
#include <stdio.h>
#include <stdlib.h>
#include "view.h"
```

⚠ Les fichiers doivent bien être dans le même répertoire, sinon ils ne se retrouveront pas. En cas de messages d'erreur, il faut bien vérifier que les fichiers sont bien placés dans le bon répertoire, qu'il n'y a pas de typo, etc...

📝 On notera bien la différence entre l'inclusion des bibliothèques `<stdio.h>` par exemple et notre fichier personnel `"view.h"` :

- Les bibliothèques inclues dans le répertoire de l'IDE sont inclues entre **chevrons** (`< >`)
- Les fichiers `.h` du projet, inclues dans le répertoire **du projet**, sont inclues entre **guillemets** (`" "`)

Et voilà comment fonctionne un vrai projet !

Les prototypes sont les modes d'emploi de la fonction pour notre ordinateur. On retient surtout que :

- Dès lors qu'on veut appeller une fonction depuis `main.c`, il faut avoir inclu le fichier `header` grâce à `#include`,
- Cette méthode permet de ne plus avoir à se soucier de où est notre fonction par rapport au `main`.

managed to compile using gcc *.c - explain how + why

H1: Comprendre La Porte Des Variables

Les variables ne sont pas toujours accessibles ! Nous allons voir dans quels cas on peut faire appel à elles.

⚠️ Il est important de limiter la portée des variables. Bien qu'il puisse être tentant de créer des variables accessibles dans toutes les fonctions de toutes les fichiers du projet (aussi nommées **variables globales**), il vaut mieux éviter cette stratégie ; ça peut causer des soucis lors de la compilation !

H2: Les variables locales

📝 Une **variable locale** est une variable déclarée au sein d'une fonction ; elle est propre à cette fonction.

On peut déclarer une variable directement dans la fonction, mais elle sera supprimée de la mémoire dès que celle-ci est terminée. Par exemple, une variable déclarée dans la fonction `bite` ne pourra pas être appellée dans la fonction `couille`, ni dans le `main` !

H2: Créer une variable globale accessible uniquement dans un fichier.

On l'a dit, il faut éviter de créer des fonction globales à l'ensemble du projet ; si on veut créer une variable globale, il vaut mieux limiter son utilisation au seul fichier où elle se trouve.

Pour déclarer une variable globale, on la positionne juste après les `#include`. Ainsi, elle est accessible à tout le projet ; pour limiter sa portée, il suffit d'ajouter le mot-clé `static` devant la variable concernée, ainsi :

```
static int resultat = 0;
```

La variable `resultat` devient alors accessible à toutes les fonctions du fichier où elle se trouve, mais pas à celles dans d'autres parties du projet.

H2: Conserver la valeur d'une variable avec `static`

Limiter la portée d'une variable n'est cependant pas la seule utilisation possible de `static` : on peut également s'en servir devant une variable on sein d'une fonction pour lui indiquer de garder en mémoire la valeur de l'opération précédente - la prochaine fois qu'on appelera la fonction, elle aura conservé sa valeur.

Par exemple :

```c
int incremente();

int main(int argc, char *argv[])
{
    printf("%d\n", incremente());
    printf("%d\n", incremente());
    printf("%d\n", incremente());
    printf("%d\n", incremente());

    return 0;
}

int incremente()
{
    static int nombre = 0;

    nombre++;
    return nombre;
}
```

📝 Explications

Ici, on a déclaré `nombre` comme variable de la fonction `incremente`. `nombre` est donc crée pour la première fois lorsqu'on appelle `incremente` ; cependant, grâce à `static`, la valeur de la variable n'est pas supprimée lorsque la fonction se termine. Elle est gardée en mémoire, et ainsi, lorsque qu'on rappelle incrémente une seconde fois, la valeur de la variable augmente de 1.

Voilà le résultat de la fonction :

```
1
2
3
4
```

Et voilà le résultat de la même fonction si on retire le `static` :

```
1
1
1
1
```

Dans le deuxième cas, la fonction incrémente bien de 1 à chaque initialisation, mais elle repart de la fonction initiale de la variable (0), car la valeur est supprimée lorsque la fonction `incremente` parvient à son terme.

H2 : Créer une fonction locale à un fichier.

Les fonctions, comme les variables, sont par défaut accessibles depuis l'ensemble du projet. On peut, si besoin, les verrouiller de la même façon qu'une variable, c'est à dire en utilisant `static`.

```
static int triple(int nombre)
{
    // Instructions
}
```

Pour que tout roule, il faut également déclarer le prototype en `static` :

`static int triple(int nombre);`

H2: En bref

Programmation modulaire

- Un programme est comporté de plusieurs fichiers `.c` ; chaque fichier `.c` a un fichier frère du même nom, mais avec une extension `.h`, qui contient les **prototypes** des **fonctions** déclarées dans les fichiers `.c`.
- Les fichiers sont liés entre eux grâce aux préprocesseurs `#include`. Lors de la compilation, ils sont transformés en fichiers `-o` binaires, qui sont ensuite liés entre eux par un linker dans un exécutable (`.exe`)

Portée des variables

- Une variable déclarée dans une fonction est dite **variable locale**. Elle n'est accessible qu'au sein de celle-ci ; sa valeur est supprimée lorsque la fonction se termine. On peut indiquer à la variable de conserver sa valeur en utilisant `static`.
- Une variable déclarée hors d'une fonction est dite **variable globale**. Elle est accessible par l'ensemble des fichiers du projet ; il faut la limiter au fichier présent en utilisant `static`, sans quoi on risque des erreurs à la compilation.
- Il en va de même pour les **fonctions**.

Failed test, wrong answers :

- Question 4 ; chose `for`, but we didn't know the numbre of iteration of the user entry ; (+ could've added "`do...` `while`)
- Question 5 : wrong symbol because im dumb ; should've chosen `!=` (different) and not `==` (egal).
- Question 7 : mistake between two `else... if` ; chose the one with no end loop.

To-do tomorrow

- [ ] Retake test
- [ ] Explain gcc *.c
- [ ] Study log + goals
- [ ] Write the game program