

Langage C

Modularité inter-fichiers



`#include<stdio.h>`

Rappel : structure générale d'un programme

Fichier main.c

```
#include librairies standard  
#define CONSTANTES SYMBOLIQUES
```

Déclaration des fonctions ;

```
int main()  
{  
appel aux fonctions;  
return 0;  
}
```

Définition des fonctions

Rappel : structure générale d'un programme

Fichier main.c

```
#include librairies standard  
#define CONSTANTES SYMBOLIQUES
```

Déclaration des fonctions ;

```
int main  
{  
appel aux fonctions;  
return 0;  
}
```

Définition des fonctions

Les fonctions ne sont utilisables que dans ce fichier

Si on veut les réutiliser ailleurs, on ne va pas faire des copier-coller ...

On va les répartir dans un ou plusieurs fichiers sources dits librairies ou bibliothèques →
Modularité inter-fichiers

Avantages de la modularité inter-fichiers

- hiérarchisation des problèmes,
- lisibilité,
- réutilisation,
- modifications centralisées,
- compilation séparée.

Principe

1. On crée un fichier source .c par thème et on l'inclut dans le projet
Code Blocks :

Fichier tableaux.c

Fichier main.c

```
#include librairies standard
#define CONSTANTES SYMBOLIQUES

/*Déclaration*/
void init_tab(int tab[],int dim) ;

int main()
{
appel aux fonctions;
return 0;
}

/*Définition*/
void init_tab(int tab[],int dim)
{code de la fonction
}
```

Principe

1. On crée un fichier source .c par thème et on l'inclut dans le projet Code Blocks :
2. On y copie-colle les définitions et les librairies utiles :

Fichier tableaux.c

```
#include <stdio.h>
/*Définition*/
void init_tab(int tab[],int dim)
{code de la fonction
}
```

Fichier main.c

```
#include librairies standard
#define CONSTANTES SYMBOLIQUES

/*Déclaration*/
void init_tab(int tab[],int dim) ;

int main
{
appel aux fonctions;
return 0;
}
```

Principe

1. On crée un fichier source .c par thème et on l'inclut dans le projet Code Blocks :
2. On y copie-colle les définitions et les librairies utiles :

Fichier tableaux.c

```
#include <stdio.h>
/*Définition*/
void init_tab(int tab[],int dim)
{code de la fonction
}
```

3. On crée un fichier d'entête .h par thème, du même nom que le fichier source :

Fichier tableaux.h

Fichier main.c

```
#include librairies standard
#define CONSTANTES SYMBOLIQUES

/*Déclaration*/
void init_tab(int tab[],int dim) ;

int main
{
appel aux fonctions;
return 0;
}
```

Principe

1. On crée un fichier source .c par thème et on l'inclut dans le projet Code Blocks :
2. On y copie-colle les définitions et les librairies utiles :

Fichier tableaux.c

```
/*Définition*/  
void init_tab(int tab[],int dim)  
{code de la fonction  
}
```

3. On crée un fichier d'entête .h par thème, du même nom que le fichier source :
4. On y copie-colle les déclarations :

Fichier tableaux.h

```
/*Déclaration*/  
void init_tab(int tab[],int  
dim) ;
```

Fichier main.c

```
#include librairies standard  
#define CONSTANTES SYMBOLIQUES
```

```
int main()  
{  
appel aux fonctions;  
return 0;  
}
```


Principe

1. On crée un fichier source .c par thème et on l'inclut dans le projet Code Blocks :
2. On y copie-colle les définitions et les librairies utiles :

Fichier tableaux.c

```
/*Définition*/  
void init_tab(int tab[],int dim)  
{code de la fonction  
}
```

3. On crée un fichier d'entête .h par thème, du même nom que le fichier source :
4. On y copie-colle les déclarations et entêtes :

Fichier tableaux.h

```
/*Déclarations et entêtes*/  
void init_tab(int tab[],int  
dim) ;
```

5. On inclut la bibliothèque dans main.c :

Fichier main.c

```
#include librairies standard  
#include "tableaux.h"  
#define CONSTANTES SYMBOLIQUES  
  
int main  
{  
appel aux fonctions;  
return 0;  
}
```

Principe

1. On crée un fichier source .c par thème et on l'inclut dans le projet Code Blocks :
2. On y copie-colle les définitions et les librairies utiles :

Fichier tableaux.c

```
/*Définition*/  
void init_tab(int tab[],int dim)  
{code de la fonction  
}
```

3. On crée un fichier d'entête .h par thème, du même nom que le fichier source :
4. On y copie-colle les déclarations et entêtes:

Fichier tableaux.h

```
/*Déclarations et entêtes*/  
void init_tab(int tab[],int  
dim) ;
```

5. On inclut la bibliothèque dans main.c :

Fichier main.c

```
#include librairies standard  
#include "tableaux.h"  
#define CONSTANTES SYMBOLIQUES  
  
int main()  
{  
appel aux fonctions définies dans  
tableaux.c;  
return 0;  
}
```

Quelques remarques

1. tableaux.c et main.c peuvent être compilés séparément : intérêt pour les gros fichiers et la centralisation des modifications et du débogage.
2. Les .h (pour header) ne sont pas des fichiers source : ils ne sont pas compilables.
3. Ce que l'on vient de voir est bien sûr valable pour toutes les autres bibliothèques de fonctions que vous allez créer.
4. Il y a une bibliothèque par thématique (tableaux, fichiers etc ...) MAIS PAS une bibliothèque par fonction !

Les #include

`#include<...>` ira chercher le .h dans les bibliothèques préinstallées.

exemple : `#include<stdio.h>`

Sinon :

- ❖ si le fichier **fichier.h** est dans le répertoire du projet courant on écrira :

`#include "fichier.h"`

- ❖ si le fichier **fichier.h** est dans un autre répertoire, on écrira le chemin d'accès complet au fichier, par exemple :

`#include "C:\\transfert\\Sylvie\\Enseignement en
C\\Enseignement en C 2013\\td8\\Fichiers pour les
élèves\\fichier.h"`

Compilation conditionnelle

Quand on ouvre un nouveau fichier .h, **tableaux.h** par exemple, CodeBlocks génère les lignes en vert ci-dessous :

```
#ifndef TABLEUX_H_INCLUDED
#define TABLEUX_H_INCLUDED
Mettre ici les prototypes des fonctions
#endif // TABLEUX_H_INCLUDED
```

TABLEAUX_H_INCLUDED est un identificateur. S'il est rencontré pour la 1^{ère} fois par le processeur, le texte situé entre #define et #endif est inclus. Sinon, il n'est pas pris en compte.

Application : protection des fichiers d'en-tête contre les inclusions multiples (cf. stdio.h par exemple)