

COURS 5

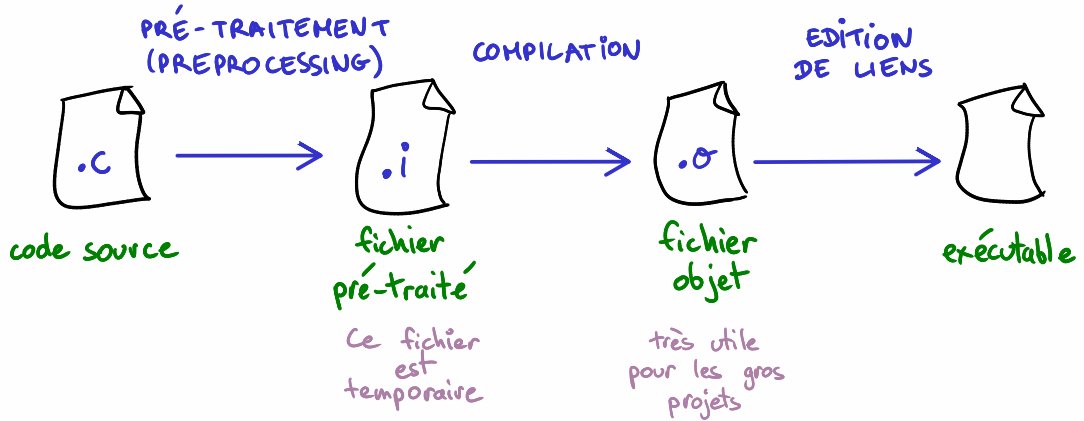
COMPILATION & BIBLIOTHÈQUES

PARTIE ①

MÉCANISME DE LA
COMPILATION

PRINCIPALES ÉTAPES DE LA COMPILATION

Que se passe-t-il quand j'écris ?
gcc code_source.c -o mon_programme_cheri



EXEMPLE

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

#define NOMBRE_REPETITIONS 18

int main(){
    for(int i = 0; i < NOMBRE_REPETITIONS; i++){
        printf("Bonjour !\n");
    }
    return EXIT_SUCCESS;
}
```



```
1236 /usr/include/stdint.h 2
3 "exemple.c" 2
7 "exemple.c"
int main(){
    for(int i = 0; i < 18; i++){
        printf("Bonjour !\n");
    }
    return
13 "exemple.c" 3 4
13 "exemple.c"
}
```



```
1236 /usr/include/stdint.h 2
3 "exemple.c" 2
7 "exemple.c"
int main(){
    for(int i = 0; i < 18; i++){
        printf("Bonjour !\n");
    }
    return
13 "exemple.c" 3 4
13 "exemple.c"
}
```



```
1236 /usr/include/stdint.h 2
3 "exemple.c" 2
7 "exemple.c"
int main(){
    for(int i = 0; i < 18; i++){
        printf("Bonjour !\n");
    }
    return
13 "exemple.c" 3 4
13 "exemple.c"
}
```

PRÉTRAITEMENT

- phase préliminaire à la compilation
- assurée en C par cpp (= préprocesseur C)
- transforme un fichier texte en un autre fichier texte

En C les directives pour le préprocesseur commencent par #

(Ex)

```
ici → #include <stdio.h>
ici → #include <stdlib.h>
ici → #define NOMBRE_REPETITIONS 18

int main(){
    for(int i = 0; i < NOMBRE_REPETITIONS; i++){
        printf("Bonjour !\n");
    }
    return EXIT_SUCCESS;
}
```

exemple.c

```
# 1026 "/usr/include/stdlib.h" 3 4
# 3 "exemple.c" 2
|
# 7 "exemple.c"
int main(){

    for(int i = 0; i < 18; i++){
        printf("Bonjour !\n");
    }

    return
# 13 "exemple.c" 3 4
    0
# 13 "exemple.c"
;
}
```

exemple.i (fin de fichier)

PRÉTRAITEMENT

Principales instructions pour le pré-traitement:

définition de
macros

inclusion de
fichiers

compilation
conditionnelle

on va voir seulement ces deux

(Ex)

```
ici → #include <stdio.h>
ici → #include <stdlib.h>
ici → #define NOMBRE_REPETITIONS 18

int main(){
    for(int i = 0; i < NOMBRE_REPETITIONS; i++){
        printf("Bonjour !\n");
    }
    return EXIT_SUCCESS;
}
```

exemple.c

```
# 1026 "/usr/include/stdlib.h" 3 4
# 3 "exemple.c" 2
|
# 7 "exemple.c"
int main(){

    for(int i = 0; i < 18; i++){
        printf("Bonjour !\n");
    }

    return
# 13 "exemple.c" 3 4
    0
# 13 "exemple.c"
;
}
```

exemple.i (fin de fichier)

PRÉTRAITEMENT : DÉFINITION DE MACROS

s'effectue avec l'instruction `#define`

SYNTAXE	<code>#define NOM-MACRO code-substitution</code>
---------	--

Le préprocesseur va remplacer toute occurrence de `NOM-MACRO` dans le code source par `code-substitution`

Remarques

- s'écrit conventionnellement en majuscules.
- utile pour la définition de constantes
- ça n'est pas une variable (c'est purement textuel)

(Ex

```
#include <stdio.h>
#include <stdlib.h>
#define TAILLE_MAX 30
```

```
int main(){
    char nom_utilisateur[TAILLE_MAX];
    printf("Quel est votre nom ? \n");
    fgets(nom_utilisateur,TAILLE_MAX,stdin);
    printf("Bonjour %s ! L'heure est ",nom_utilisateur);
    printf(__TIME__);
    printf(".\n");
    return EXIT_SUCCESS;
}
```

← certaines macros sont prédéfinies



```
int main(){
    char nom_utilisateur[30];
    printf("Quel est votre nom ? \n");
    fgets(nom_utilisateur,30,
# 8 "repete_nom.c" 3 4
                                stdin
# 8 "repete_nom.c"
                                );
    printf("Bonjour %s ! L'heure est ",nom_utilisateur);
    printf("11:44:27");
    printf(".\n");

    return
# 13 "repete_nom.c" 3 4
    0
# 13 "repete_nom.c"
    ;
}
```

On peut également faire des macros paramétrées mais interdit pour ce cours.

PRÉTRAITEMENT : INCLUSION DE FICHIERS

→ s'effectue avec l'instruction `#include`

→ permet d'inclure le contenu d'un fichier dans le fichier courant

→ On n'utilisera ça que pour inclure des ".h" (des entêtes)

↑
on reviendra sur
cette notion

SYNTAXE	<code>#include < .h ></code>	(fichier fourni par le compilateur)
	<code>#include " .h "</code>	(fichier dans le répertoire courant)

Principal intérêt : utiliser des fonctions ou des macros provenant d'autres fichiers

Par ex, `printf` vient de `stdio.h`
`EXIT_SUCCESS` vient de `stdlib.h`

transforme le fichier texte en code machine
(le fichier prétraité) (le fichier objet)

example.i

processus très compliqué

example.s

[illegible]

example. σ

2) FICHIER OBJET

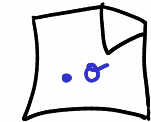
fichier binaire qui encode uniquement les fonctions du fichier de base.

Par ex, le code de printf n'est pas inclus dans le fichier objet

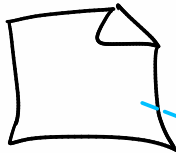
ÉDITION DE LIENS (OU LINKING)

L'éditeur de liens (ou linker en anglais) permet de produire un exécutable à partir de plusieurs fichiers objets/bibliothèques

Dans le cas où il y a un unique fichier objet, ^{comme printf} l'éditeur de liens va permettre d'utiliser des fonctions "hors exécutable" via des bibliothèques dynamiques partagées



FICHER
OBJET



EXÉCUTABLE

dll sous windows
(dynamic linked library)



BIBLIOTHÈQUE
PARTAGÉE

LES OPTIONS UTILES DE GCC

OPTION	Description
-o nom_executable	génère l'exécutable et le baptise nom_executable (par défaut c'est a.out)
-c	génère le fichier objet
-save-temps	génère tous les fichiers intermédiaires lors de la compilation
-Wall	affiche tous les avertissements possibles.
-Werror	les avertissements sont considérés comme des erreurs
-lm	permet d'utiliser la bibliothèque <math.h>

PARTIE ~~II~~

5 BIBLIOTHÈQUES
STANDARDS
À CONNAÎTRE

MATH. H

regroupe toutes les fonctions mathématiques dont vous pouvez rêver

Pêle-mêle: \cos , \arccos , \sin , \arcsin , \tan , \arctan , $\sqrt{}$, \log_2 , pow ...
 racine carrée, log en base 2, puissance

Mais aussi...

M_PI nombre π (macro)

INFINITY	nombre flottant infini (macro)
----------	--------------------------------



Il faut rajouter l'option `-lm` quand on lance gcc pour qu'il puisse inclure `math.h`

BIBLIOTHÈQUE

STRINGS.H

manipule les chaînes de caractères

strlen(ch)	Renvoie la longueur de la chaîne de caractères
strcpy(cible,source)	Copie une chaîne de caractères dans un tableau
strcmp(ch1,ch2)	Compare 2 chaînes de caractères lexicographiquement
strchr(ch,c)	Détermine si un caractère apparaît dans une chaîne de caractères

BIBLIOTHÈQUE STUDIO.H

gère les entrées/sorties du programme $I = in$
 $O = out$

Vous connaissez déjà printf, scanf, fgets ...

Permet aussi la manipulation des fichiers (sujet d'un futur TP?)

BIBLIOTHÈQUE TIME.H

Manipulation des dates et du temps

time(NULL)	renvoie l'écart de temps entre maintenant et le 1 ^{er} janvier 1970 minuit ("timestamp")
difftime(fin,debut)	renvoie la différence de deux timestamps
clock()	renvoie le nombre de tics écoulés depuis le début du programme
CLOCKS_PER_SEC	nombre de tics par seconde (macro)

BIBLIOTHÈQUE

STDLIB.H

bibliothèque multi-utilitaire

**SORTIR
DU PROGRAMME**

je vais vous en
parler au tableau!



**NOMBRES
PSEUDO
ALÉATOIRES**

exit(code)	fonction qui prend en paramètre un code de sortie et qui arrête le programme
EXIT_SUCCESS	code de sortie sans erreur (macro)
EXIT_FAILURE	code de sortie avec erreur (macro)

rand()	renvoie un entier pseudo-aléatoire ≥ 0
srand(graine)	initialise une graine aléatoire
RAND_MAX	taille max pour un entier pseudo aléatoire (macro)

abs	valeur absolue d'un entier
-----	----------------------------

mais aussi...

**CONVERSION NOMBRE
↔ CHAÎNE DE CARACTÈRES**

**ALLOCATION
DYNAMIQUE
DE LA MÉMOIRE**

← on verra ça
la semaine
prochaine!

COMMENT GÉNÉRER DES NOMBRES ALÉATOIRES?

En programmation, rien n'est aléatoire...

Mais alors comment on fait?

On part d'un nombre
appelé **graine**

18 

Toute l'aléatoire
provient
d'une graine

On applique une formule
mathématique compliquée



Ce nombre semble aléatoire :
il est pseudo-aléatoire

1804289383

Pour calculer le nombre suivant,
on réapplique la formule
compliquée au nombre
pseudo-aléatoire suivant



846930886



Pour changer
les nombres
pseudo-aléatoires,
on change la
graine
(par exemple en
fonction de l'heure)

En C :

```
srand(time(NULL));
```

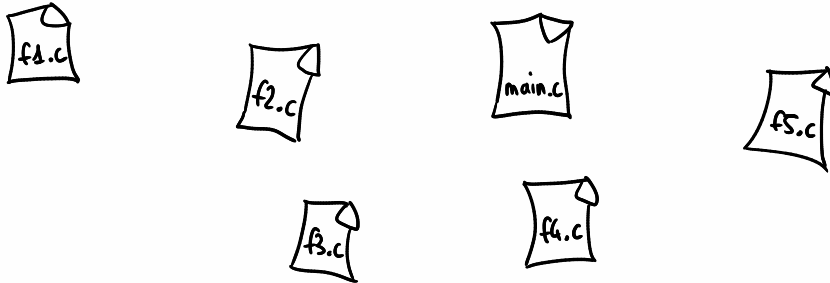
PARTIE ~~III~~

GESTION DE
MULTIPLES FICHIERS

INITIATION À LA PROGRAMMATION MODULAIRE

Dans les gros projets, il est impensable de tout coder dans un seul fichier code .

On va partager le code dans plusieurs fichiers code = les modules



On parle de programmation modulaire.

Comment s'organiser ?

LES FICHIERS ENTÊTE

entête = header en anglais = fameux fichiers .h

Un fichier .c et un fichier .h forment un module

- fichier .c = contenu du code source
 - fichier .h = résumé de ce que contient le fichier .c
- ≈ mode d'emploi pour le codeur et le compilateur

généralement
on inclut
l'entête →

```
#include "numeration.h"

int nombre_chiffres(int n){
    int cpt = 1;
    if (n < 0) { n = -n; }
    while ( n >= BASE ){
        n = n/BASE;
        cpt++;
    }
    return cpt;
}
```

numeration.c

```
#define BASE 10

int nombre_chiffres(int nb);
/** Renvoie le nombre de chiffres
 * selon la base de numération
 * donnée par BASE (défaut : 10)
 * @param un nombre entier
 * @return le nombre de chiffres de
 * de ce nombre entier */
```

numeration.h

Un fichier entête regroupe :

- les prototypes des fonctions
- les descriptions des fonctions
- les définitions des constantes
- et des trucs hors programme du cours

← KÉZAKO?

(Ex)

il n'y
a pas
de main!

LES PROTOTYPES DE FONCTIONS

permet de déclarer les fonctions

SYNTAXE

type_sortie nom_fonction (type1 param1, type2 param2,...);

en gros comme une définition de fonctions mais où remplacerait les accolades + ce qu'il y a à l'intérieur par un point virgule

Exemples

```
int nombre_chiffres(int nb);  
int chaine_vers_nombre(char* ch);  
float puissance(float nb, int exposant);
```

L'intérêt est double :

- Pour le codeur. Indique de manière concise comment s'utilise une fonction
- Pour le compilateur. On peut utiliser des fonctions qui sont incluses dans d'autres fichiers sans qu'elles soient définies dans le fichier actuel.

Ex:

```
int inconnue(int n);  
int une_valeur(){  
    return inconnue(6);  
}
```

compile*!

*: en un fichier objet

LES DESCRIPTIONS

- sert uniquement comme documentation
- plusieurs normes (ici Doxygen)

(Ex

```
float puissance(float nb, int exposant);  
/** Calcule l'exponentiation d'un flottant par un entier  
@param nb, un nombre (de type float) sur lequel on va  
appliquer la puissance  
@param exposant, un entier positif de type int  
@return nb à la puissance exposant **/
```

UN EXEMPLE JOUET

MODULE 1

```
int double(int nb);
```

fois2.h

```
#include "fois2.h"
int double(int nb){
    return 2*nb;
}
```

fois2.c

MODULE 2

```
void affiche_double(int nb);
```

affiche.h

```
#include "affiche.h"
#include "fois2.h"
#include <stdio.h>
void affiche_double(int nb){
    printf("%d",double(nb));
}
```

affiche.c

PROGRAMME PRINCIPAL

```
#include "affiche.h"
#include <stdlib.h>
int main(){
    affiche_double(10);
    return EXIT_SUCCESS;
}
```

main.c

Comment compiler main.c?

✗ gcc main.c -o affiche20 ne marche pas car affiche_double n'est pas définie

✓ gcc main.c affiche.c fois2.c -o affiche20 fonctionne

COMPILATION SÉPARÉE

MODULE 1

```
int double(int nb);
```

fois2.h

```
#include "fois2.h"
int double(int nb){
    return 2*nb;
}
```

fois2.c

MODULE 2

```
void affiche_double(int nb);
```

affiche.h

```
#include "affiche.h"
#include "fois2.h"
#include <stdio.h>
void affiche_double(int nb){
    printf("%d",double(nb));
}
```

affiche.c

PROGRAMME PRINCIPAL

```
#include "affiche.h"
#include <stdlib.h>
int main(){
    affiche_double(10);
    return EXIT_SUCCESS;
}
```

main.c

Pourquoi `gcc main.c affiche.c fois2.c -o affiche20` marche ?

PRETRAITEMENT

fois2.c  → fois2.i 

affiche.c  → affiche.i 

main.c  → main.i 

COMPILEUR SÉPARÉE

MODULE 1

```
int double(int nb);
```

fois2.h

```
int double(int nb);  
int double(int nb){  
    return 2*nb;  
}
```

fois2.i

MODULE 2

```
void affiche_double(int nb);
```

affiche.h

```
void affiche_double(int nb);  
int double(int nb);  
<contenu de stdio.h>  
void affiche_double(int nb){  
    printf("%d",double(nb));  
}
```

affiche.i

PROGRAMME PRINCIPAL

```
void affiche_double(int nb);  
<contenu de stdlib.h>  
int main(){  
    affiche_double(10);  
    return EXIT_SUCCESS;  
}
```

main.i

Pourquoi `gcc main.c affiche.c fois2.c -o affiche20` marche ?

PRETRAITEMENT

fois2.c  → fois2.i 

affiche.c  → affiche.i 

main.c  → main.i 

COMPILEUR SÉPARÉE

MODULE 1

```
int double(int nb);
```

fois2.h

```
int double(int nb);  
int double(int nb){  
    return 2*nb;  
}
```

fois2.i

MODULE 2

```
void affiche_double(int nb);
```

affiche.h

```
void affiche_double(int nb);  
int double(int nb);  
<contenu de stdio.h>  
void affiche_double(int nb){  
    printf("%d",double(nb));  
}
```

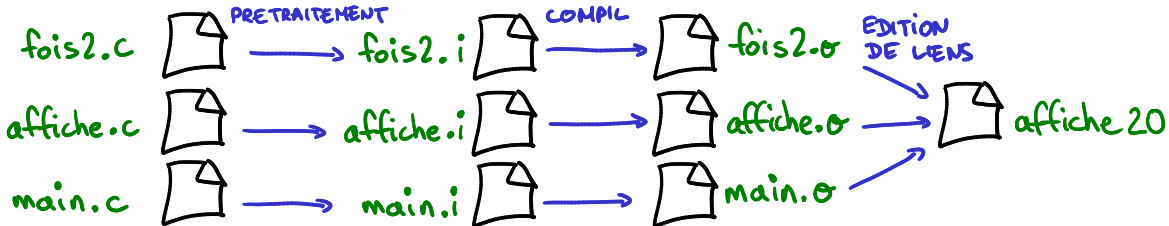
affiche.i

PROGRAMME PRINCIPAL

```
void affiche_double(int nb);  
<contenu de stlib.h>  
int main(){  
    affiche_double(10);  
    return EXIT_SUCCESS;  
}
```

main.i

Pourquoi `gcc main.c affiche.c fois2.c -o affiche20` marche ?



Pb: Ds les gros projets, recompiler tous les modules peut être très long

COMPILEMENT SÉPARÉE

MODULE 1

```
int double(int nb);
```

fois2.h

```
int double(int nb);  
int double(int nb){  
    return 2*nb;  
}
```

fois2.i

MODULE 2

```
void affiche_double(int nb);
```

affiche.h

```
void affiche_double(int nb);  
int double(int nb);  
<contenu de stdio.h>  
void affiche_double(int nb){  
    printf("%d",double(nb));  
}
```

affiche.i

PROGRAMME PRINCIPAL

```
void affiche_double(int nb);  
<contenu de stlib.h>  
int main(){  
    affiche_double(10);  
    return EXIT_SUCCESS;  
}
```




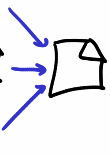
main.i

Pour économiser du temps, on compile de manière séparée les fichiers .c

fois2.c  →  fois2.o gcc fois2.c -c

affiche.c  →  affiche.o gcc affiche.c -c

main.c  →  main.o gcc main.c -c

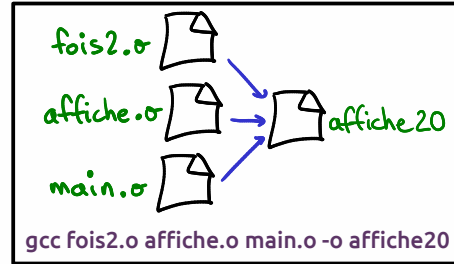
fois2.o 
affiche.o 
main.o 
 affiche20
gcc fois2.o affiche.o main.o -o affiche20

COMPILATION SÉPARÉE

fois2.c  →  fois2.o gcc fois2.c -c

affiche.c  →  affiche.o gcc affiche.c -c

main.c  →  main.o gcc main.c -c



Si on a fait une erreur dans main.c ,
pas besoin de tout recompiler.

Il faut faire :
gcc main.c -c
puis gcc fois2.o affiche.o main.o -o affiche20

C'est pas un peu lourd de faire ça manuellement ?
Si. C'est pour ça qu'on va automatiser cela.

Avec make.

MAKE

On écrit ce qu'il faut faire dans un fichier appelé **makefile**
Ce fichier est composé de règles de la forme

Syntaxe	<code>nom-regle: prerequis1 prerequis2 ...</code>  <code>commande à exécuter</code> <i>tabulation</i>
---------	--

- Il faudra ensuite écrire **make** dans un terminal pour lancer la compilation
- Les règles ne s'effectueront que si un des prérequis a changé.
- On peut exécuter une règle précise en écrivant `make <regle>`
Par ex, `make clean` supprime les fichiers .o (c'est une règle classique)

```
all: affiche20
    @echo "Compilation terminée"
affiche20: fois2.o affiche.o main.o
    gcc fois2.o affiche.o main.o -o affiche20
fois2.o: fois2.c
    gcc fois2.c -c
affiche.o: affiche.c
    gcc affiche.c -c
main.o: main.c
    gcc main.c -c
clean:
    rm fois2.o affiche.o main.o
```

makefile