Compilation et Fichiers

thibaut.lust@lip6.fr

Polytech Sorbonne

2020

`https://moodle-sciences.upmc.fr` (cours Informatique Générale EPU-R5-IGE)

Cours basé sur les diapositives créées par Julien Brajard

Plan du cours

La fonction `main`

- Un code C doit contenir obligatoirement une fonction `main`.
- Le `main` est le point d'entrée de l'exécutable.

Il est possible de communiquer des informations à l'exécutable.

Exemple : le programme `gedit` est un exécutable, on peut lui passer en argument le nom du fichier à ouvrir.

Ouvre l'éditeur de texte avec un fichier vide :

```
>> gedit
```

Ouvre le fichier `hello.c` :

```
>> gedit hello.c
```

Arguments du `main`

Il est possible de récupérer les arguments passés à l'exécutable grâce aux arguments du main.

```
int main (int argc, char *argv[]){
```

- `argc` : contient le nombre d'arguments passés à l'exécutable (nombre de mots dans la ligne de commande).
- `argv` : tableau des arguments.
  Le premier argument est le nom de l'exécutable.

Comment ça marche ?

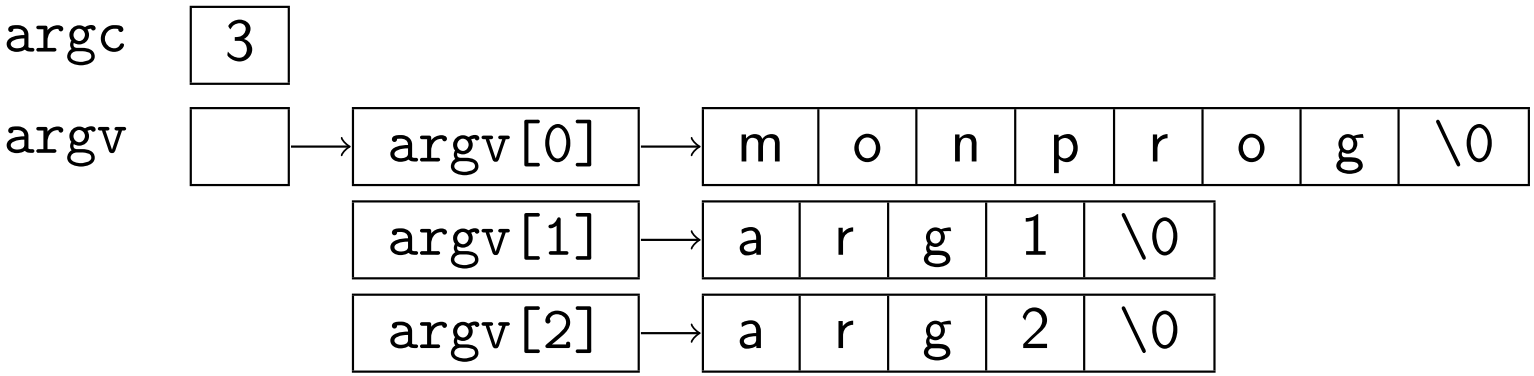
- Dans le terminal :

```
>> monprog arg1 arg2
```

- Dans le code :

```
int main (int argc, char *argv[]){
```

- En mémoire :



Un exemple

monprog.c

```c
int main (int argc, char * argv[]) {
 int i;
 for (i=1;i<argc;i++)
 {
  printf("Argument %d : %s\n",i,argv[i]);
 }
 return 0;
}
```

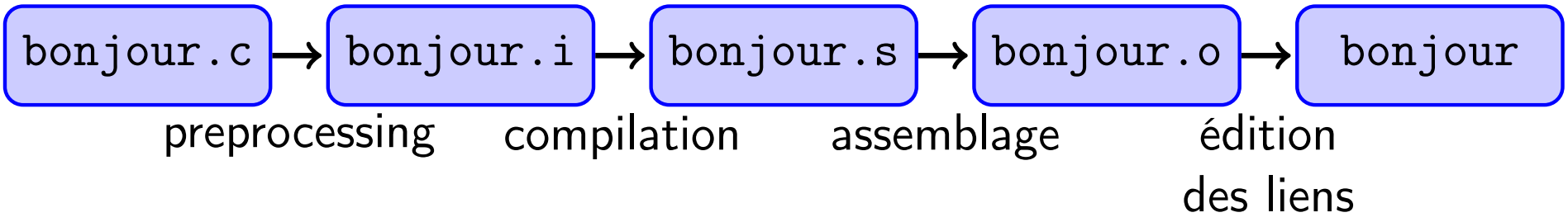
Test d'exécution

```
>> monprog fic1.txt 100
Argument 1 : fic1.txt
Argument 2 : 100
```

Utilisation

- Permet de transmettre des informations du shell au programme.
- Utile dans les scripts shell qui appellent plusieurs exécutables.
- Permet de passer certains paramètres à vos exécutables (nom de fichiers, identifiants, etc.)

Les 4 étapes de la compilation



bonjour.c

```c
#include <stdio.h>

int main () {
  // Affiche "bonjour"
  printf("bonjour\n");
}
```

→

Terminal

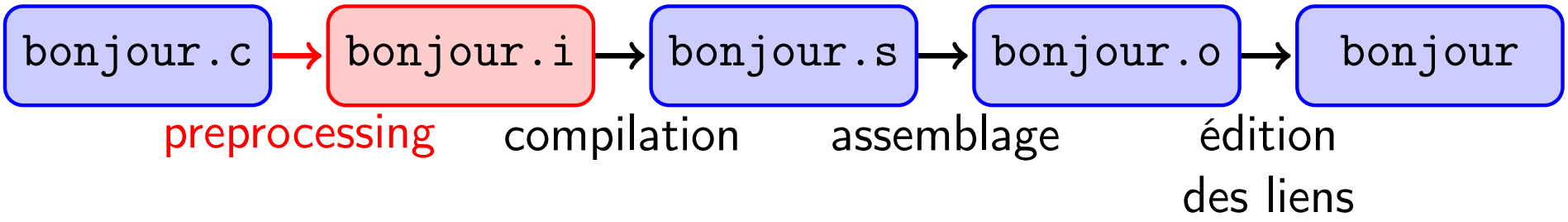
```
»gcc bonjour.c -o bonjour
```

↓

Fichier Exécutable `bonjour`

Par défaut gcc effectue les 4 étapes.

Les 4 étapes de la compilation

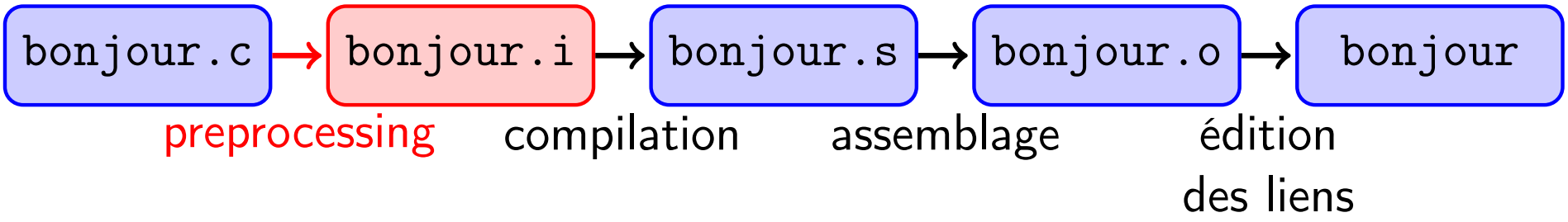


Preprocessing

Le préprocesseur effectue différentes opérations de substitution et de supression dans le code :

- Supression des commentaires (`//` et `/* */`) qui sont utiles au programmeur, mais inutiles pour le processeur.
- Inclusion des fichiers `.h` dans le fichier `.c` (directive `#include`). Ici, il permet de donner le prototype de la fonction `printf` (son format).
- Traitement des directives de compilation qui commencent par un caractère `#` (voir plus loin).

Les 4 étapes de la compilation



bonjour.c

```c
#include <stdio.h>

int main () {
  // Affiche "bonjour"
  printf("bonjour\n");
}
```

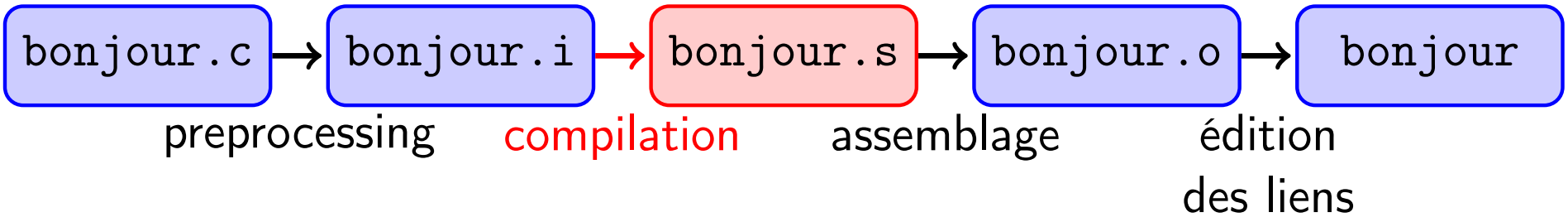
Terminal

```
»gcc –E Bonjour.c > Bonjour.i
```

bonjour.i

```c
(...) extern int printf
  (__const char *__restrict__format, ...);
(...)
# 2 "bonjour.c" 2
int main () {
  printf("bonjour\n");
}
```

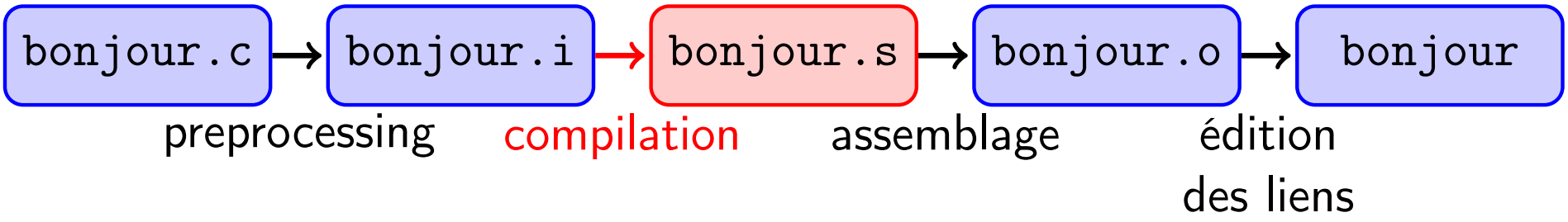
Les 4 étapes de la compilation



La compilation

La compilation (au sens strict) tranforme le langage C en assembleur.

Les 4 étapes de la compilation



bonjour.c

```c
#include <stdio.h>

int main () {
 // Affiche "bonjour"
 printf("bonjour\n");
}
```

Terminal

```
»gcc –S Bonjour.c –o bonjour.s
```

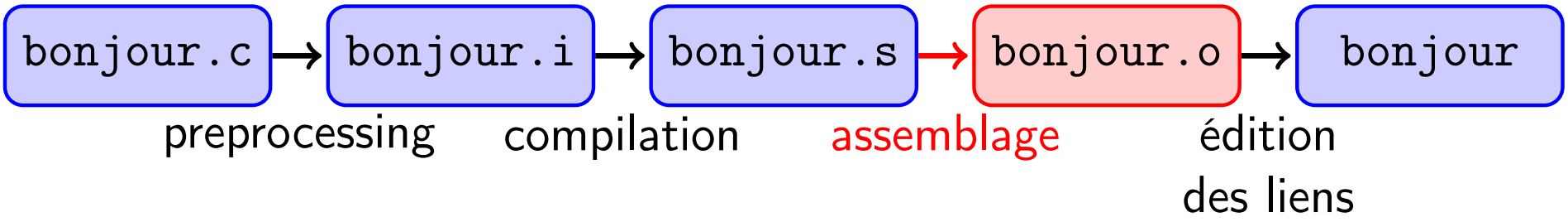
bonjour.s

```
        .file   "bonjour.c"
        .section        .rodata
.LC0:
        .string "bonjour"
(...)
        movl    $.LC0, %edi
        call    puts
(...)
        .section        .note.GNU-stack,"",
```

bonjour.s complet !

```
        .file   "bonjour.c"
        .section        .rodata
.LC0:
        .string "bonjour"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

Les 4 étapes de la compilation



Le code assembleur (encore lisible) est transformé en code machine binaire.

bonjour.s

```
        .file   "bonjour
        .section
.LC0:
        .string "bonjour"
(...)
        movl    $.LC0, %
        call    puts
(...)
        .section
 .note.GNU-stack,"",
 @progbits
```

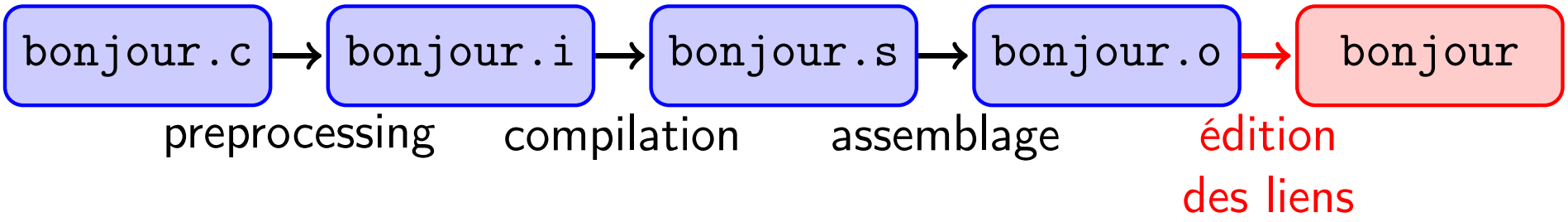
Terminal

```
»gcc -c bonjour.s
»od -x bonjour.o
```

bonjour.o

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000020 0001 003e 0001 0000 0000 0000 0000 0000
0000040 0000 0000 0000 0000 0128 0000 0000 0000
...
```

Les 4 étapes de la compilation



Le fichier `bonjour.o` est incomplet, il manque le code correspondant aux fonctions des bibliothèques (ici : la fonction `printf` de la bibliothèque `stdio.h`).

bonjour.o

(code binaire)

→

Terminal

```
»gcc bonjour.o -o bonjour
```

↓

bonjour

Fichier exécutable

La directive #define

- Déclaration de constantes (ou d'une expression fixe quelconque) : `#define` *identificateur reste_de_la_ligne*
- Lorsque le préprocesseur lit une ligne de ce type, il remplace toutes les occurences suivantes de *identificateur* dans le fichier texte par *reste_de_la_ligne*
- Par convention, on écrit l'identificateur en MAJUSCULE.

exemple.c

```
#define PI 3.14159

int main()
{
  int x;
  x = PI*2 ;
}
```

Terminal

```
» gcc -E exemple.c
  > exemple.i
```

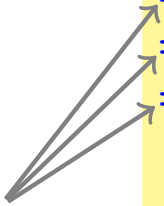
exemple.i

```
# 2 "exemple.c" 2

int main()
{
  int x;
  x = 3.14159*2 ;
}
```

Conseils sur le #define

- Il est fortement recommandé de placer les #define au début de votre programme au même endroit que les #include.
- Limitez l'utilisation des #define à des options de compilation ou à des constantes valables et utilisées dans tout le programme.

Placer ici les #define de vos programmes.



```c
#include <stdio.h>
#define PI 3.14159
#define NMAX 500
#define DEBUG_MODE

int main()
{
...
}
```

Les directives pour le préprocesseur

- `#define` *`identificateur reste_de_la_ligne`*
  Remplace *`identificateur`* par *`reste_de_la_ligne`* jusqu'à la fin du programme ou jusqu'à une instruction `#undef`.

- `#undef` *`identificateur`*
  Marque la fin du remplacement systématique initié par `#define`

- `#ifdef` *`identificateur`* `...` `#endif`
  Inclus la partie de programme située entre `#ifdef` et `#endif` si l'identificateur a été déclaré avant dans un `#define`. Sinon, la partie de programme concernée est supprimée avant compilation.

Exemple

exemple.c → **exemple.i**

```c
#define PI 3.14159

int main()
{
   int x;
   x = PI*2;
#ifdef PI
   printf("Constante PI définie");
#endif
   printf("x=%f",x);
#undef PI
#ifdef PI
   printf("Ce printf sera supprimé
par le préprocesseur");
#endif
}
```

```c
#2 exemple.c 2

int main()
{
   int x;
   x = PI*2;

   printf("Constante PI définie");

   printf("x=%f",x);









}
```

Etat des lieux

- Vos programmes sont de plus en plus gros.
- Certaines fonctionnalités ne sont pas spécifiques à un seul programme (exemple fonctions d'E/S).
- Le maintien et la compréhension des programmes est difficile.
- Le travail collaboratif est presque impossible.
- La compilation de tout le projet est nécessaire à chaque modification (même minime).

Une solution : la programmation modulaire

Possibilité de répartir un programme sur plusieurs fichiers réunissant les fonctionnalités et définitions d'un aspect particulier du programme.

Chacun de ces découpages est appelé un module.
Ex :

- Un module d'E/S,
- Un module pour les calculs,
- Un module contenant le `main()`.

$\rightarrow$ nécessite une réflexion sur le découpage de votre code.

Que contient un module ?

Un module est composé :

- Un fichier entête `.h`
- Un fichier source `.c`

- Le fichier entête décrit l'interface du module.
- Le fichier source contient l'implémentation (la définition) des fonctions.

Le fichier entête est inclus par la directive :

```
#include "entete.h"
```

Exemple : Les matrices

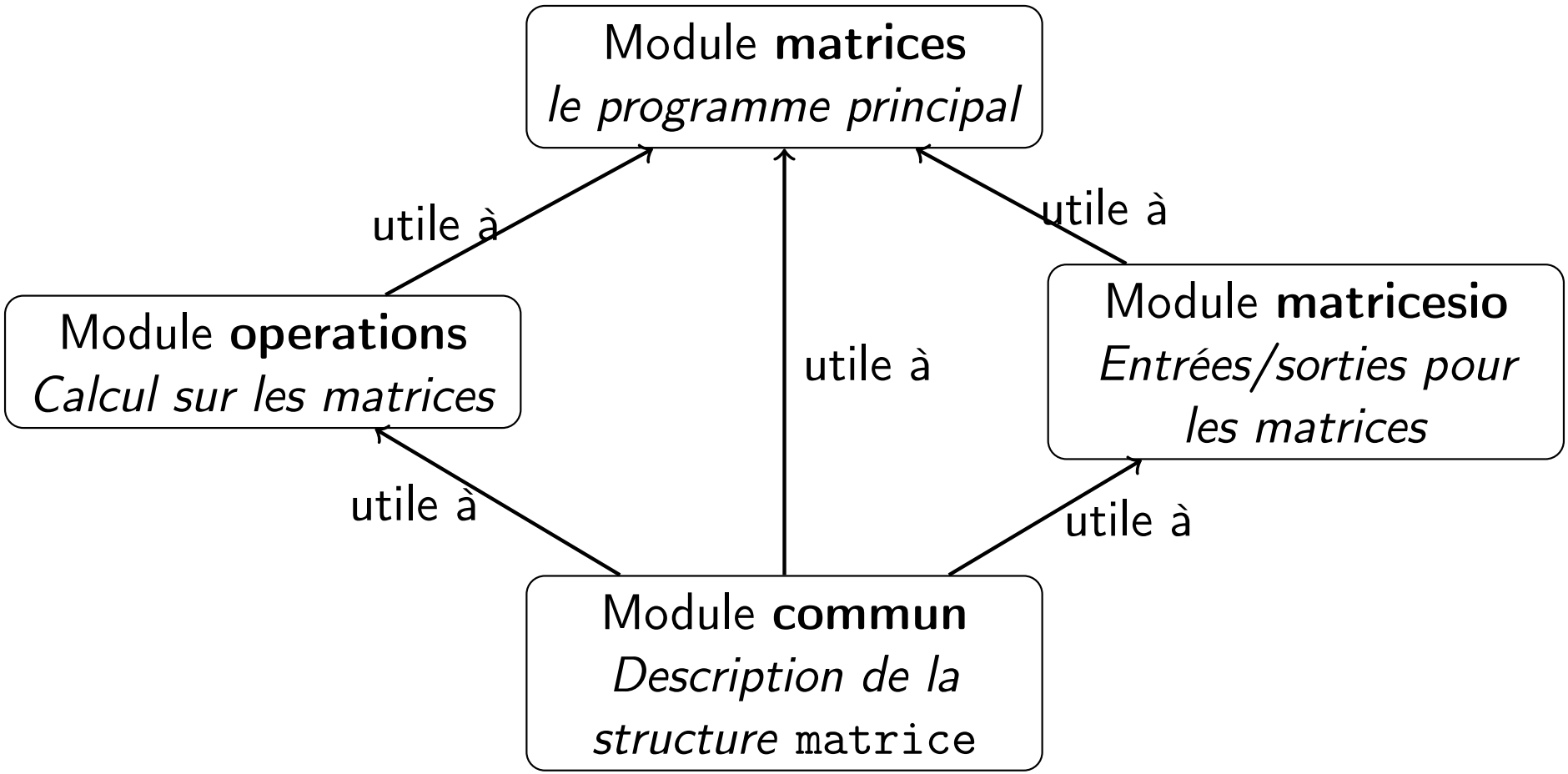
Module **matrices**
le programme principal

Module **operations**
Calcul sur les matrices

Module **matricesio**
Entrées/sorties pour les matrices

Module **commun**
Description de la structure `matrice`

Exemple : Les matrices



Le fichier entête (header) .h

Le fichier entête contient :

- Des directives `#include`
- Des directives `#define`
- Les prototypes des fonctions du module utilisables par les autres modules
- Des déclarations de variables globales
- Des modèles de structure.

Eviter de définir des variables (initialisation, etc.)

Module commun

commun.h

```c
/* Description de la structure
matrice */
#ifndef COMMUN_H
#define COMMUN_H

struct matrice {
    int col;
    int lig;
    float** mat;
};

#endif
```

pas de commun.c

(pas de fonctions)

Module matricesio

matricesio.h

```c
/* Entrées-sorties pour
  les matrices */
#include "commun.h"


struct matrice* saisir();
void
afficher(struct matrice* mat);
struct matrice* mat_uni();
```

matricesio.c

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

struct matrice* saisir(){
...
}

void afficher(struct matrice*
mat){
...
}

struct matrice* mat_uni(){
...
}
```

Module operations

operations.h

```c
/* calculs sur les matrices */
#include "commun.h"

struct matrice* add(struct matrice* m1, struct matrice* m2);
struct matrice* mul(struct matrice* m1, struct matrice* m2);
struct matrice* mul_scal(struct matrice* m, float mu);
```

operations.c

```c
#include "operations.h"

struct matrice* add(struct matrice* m1, struct matrice* m2){
...
}

struct matrice* mul(struct matrice* m1, struct matrice* m2){
...
}

struct matrice* mul_scal(struct matrice* m, float mu){
...
}
```

Module matrices

Programme principal

matrices.c

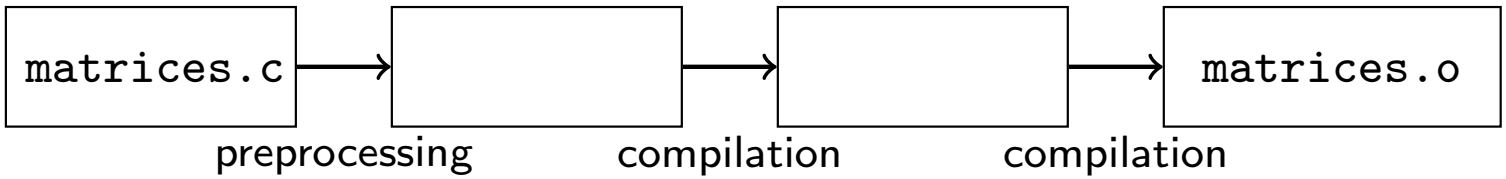
```c
#include "commun.h"
#include "matricesio.h"
#include "operations.h"

int main(){
 struct matrice *m1, *m2, *m3, *m4, *m5, *m6;
 m1 = saisir();
 m2 = saisir();
 m3 = mat_uni();
 m4 = add(m1,m2);
 afficher(m4);
 m5 = mul(m2,m3);
 afficher(m5);
 m6 = mul_scal(m4,4);
 afficher(m6);
 return 0;
}
```

matrices.h

Pas de matrices.h

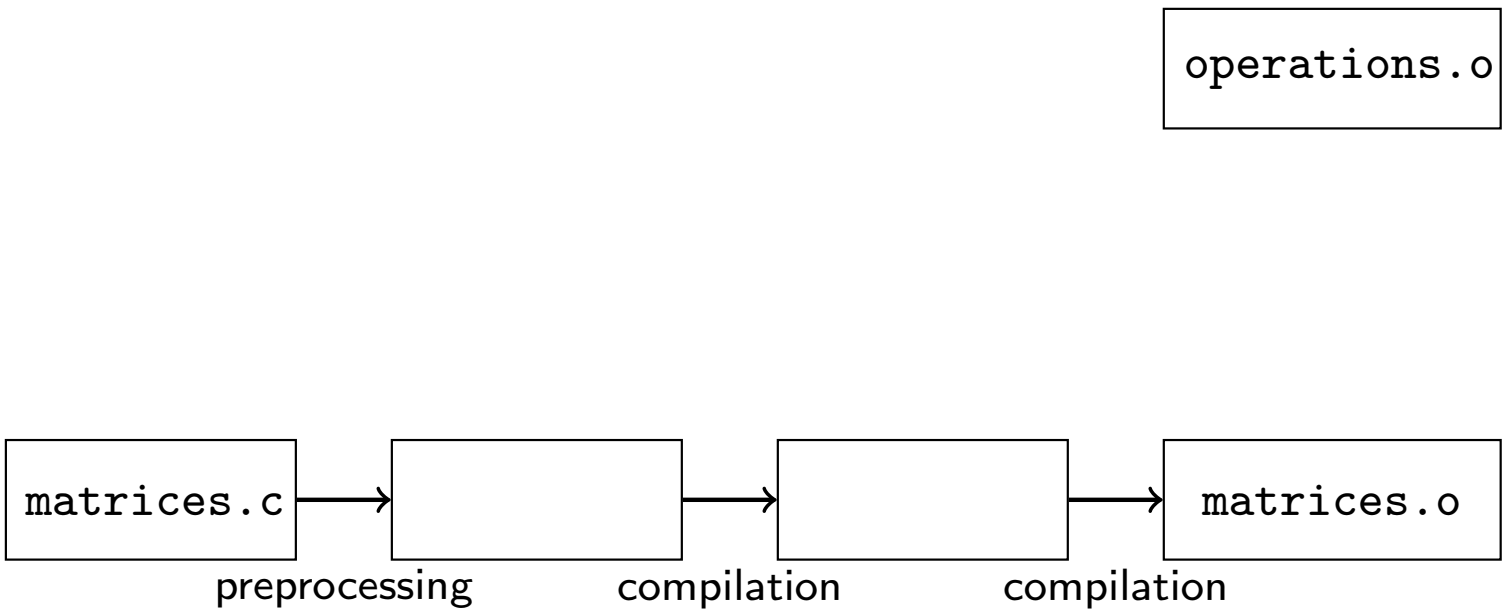
Compilation séparée

```
gcc -c matrices.c
```



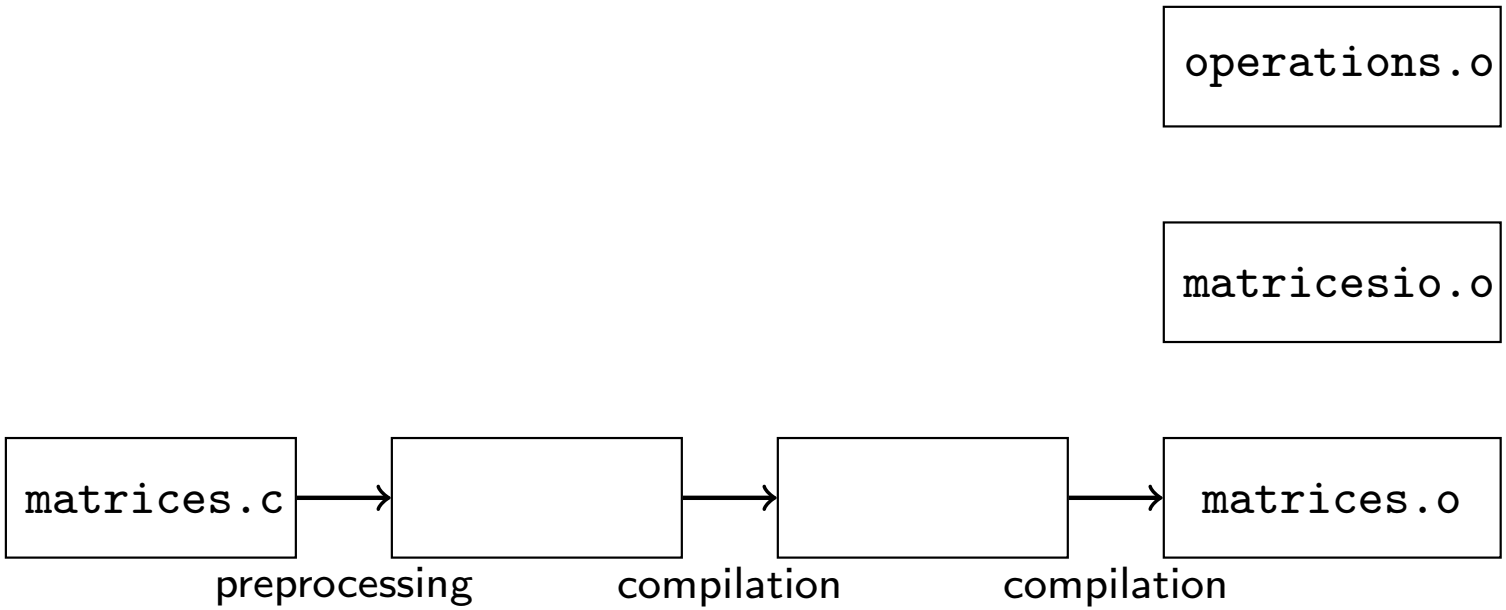
Compilation séparée

```
gcc -c matrices.c
gcc -c operations.c
```



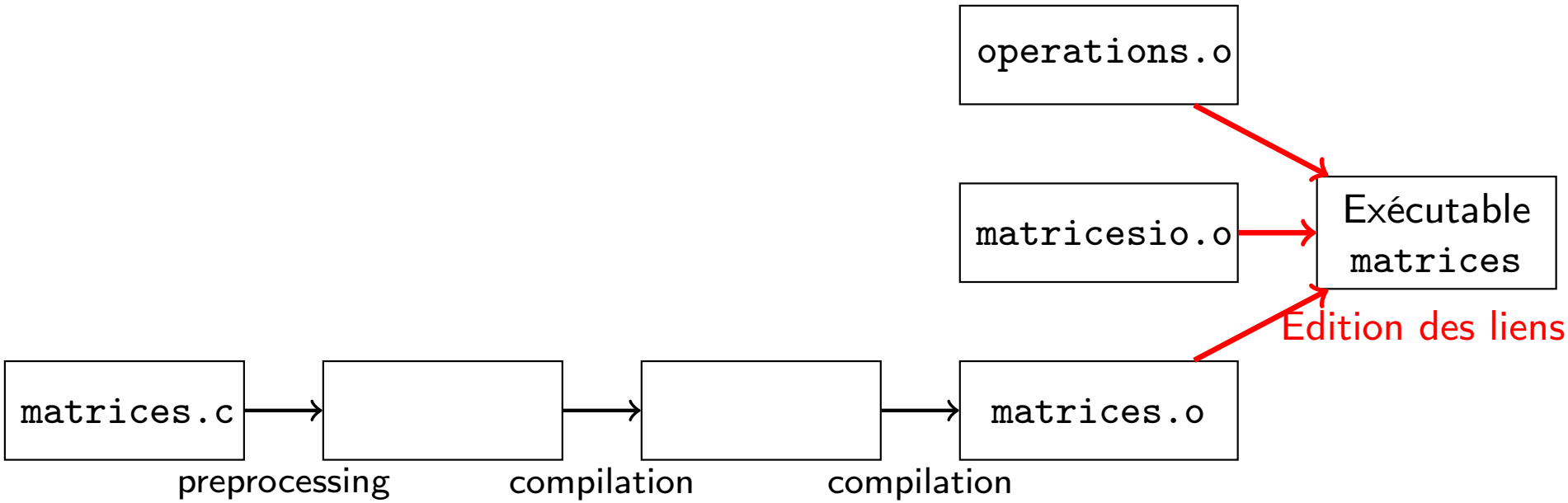
Compilation séparée

```
gcc -c matrices.c
gcc -c operations.c
gcc -c matricesio.c
```

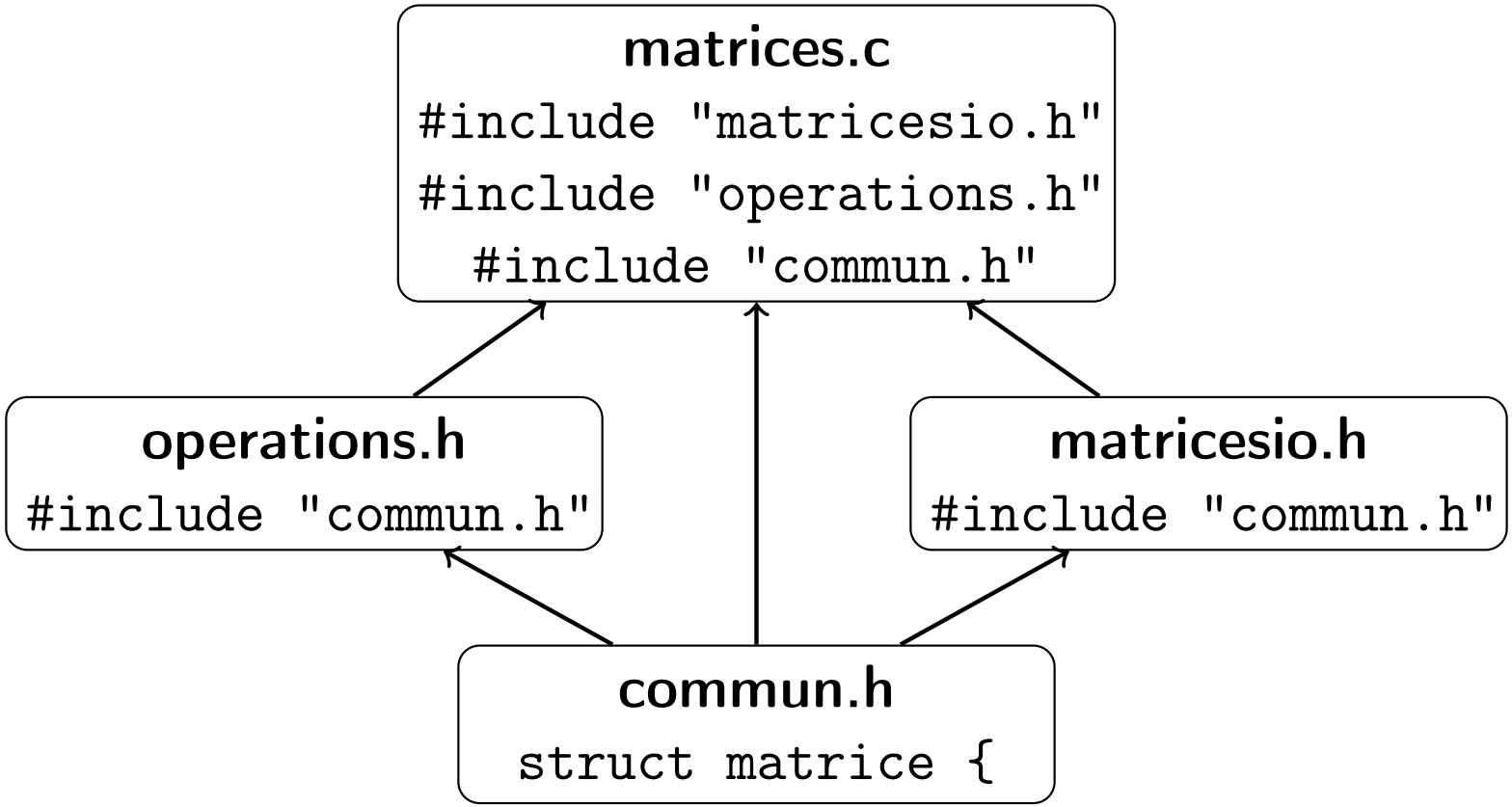


Compilation séparée

```
gcc -c matrices.c
gcc -c operations.c
gcc -c matricesio.c
gcc matrices.o matricesio.o operations.o -o matrices
```



Problème des inclusions multiples



La structure matrice est définie 3 fois !

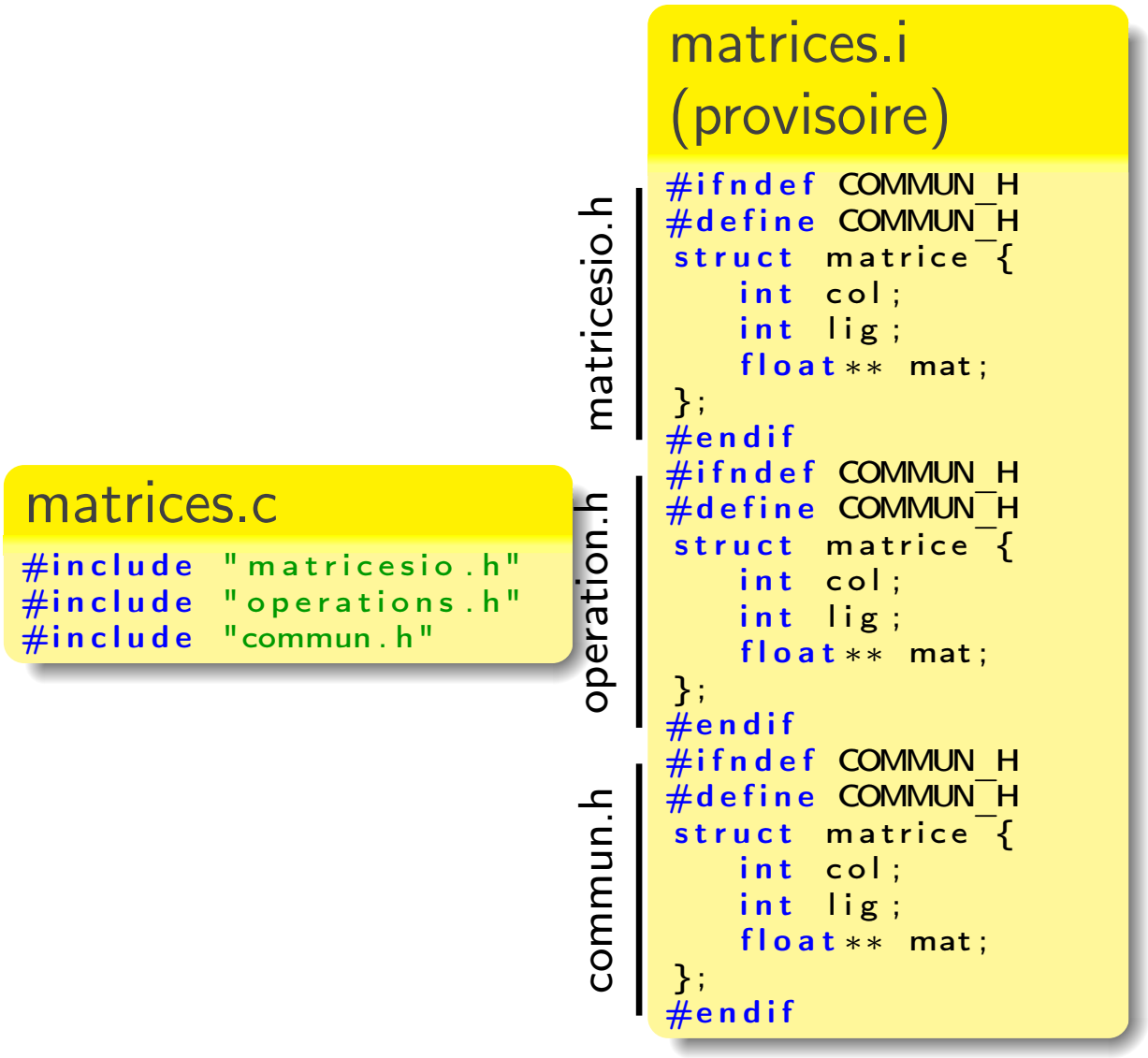
Risque d'erreur : `previous declaration of matrice`

Problème des inclusions multiples

matrices.c

```
#include "matricesio.h"
#include "operations.h"
#include "commun.h"
```

Problème des inclusions multiples



Problème des inclusions multiples

matrices.c

```c
#include "matricesio.h"
#include "operations.h"
#include "commun.h"
```

matricesio.h

operation.h

commun.h

matrices.i (provisoire)

```c
#ifndef COMMUN_H
#define COMMUN_H
struct matrice {
    int col;
    int lig;
    float** mat;
};
#endif
#ifndef COMMUN_H
#define COMMUN_H
struct matrice {
    int col;
    int lig;
    float** mat;
};
#endif
#ifndef COMMUN_H
#define COMMUN_H
struct matrice {
    int col;
    int lig;
    float** mat;
};
#endif
```

matrices.i (finale)

```c
#ifndef COMMUN_H
#define COMMUN_H
struct matrice {
    int col;
    int lig;
    float** mat;
};
#endif
```

L'outil `make`

L'outil `make` est un programme présent sous Linux permettant de déclencher un certain nombre de commandes `shell` à partir d'un fichier nommé `Makefile` ou `makefile`.

`make` permet dans votre cas d'automatiser la compilation en exploitant les avantages de la compilation séparée.
Procédez comme suit :

1. Définir les règles de compilation dans le fichier `Makefile`
2. Au moment de la compilation, invocation en entrant dans le terminal : `make`
3. La compilation se fait, en recompilant uniquement les fichiers nécessaires (ceux qui ont été modifiés depuis la dernière compilation, que l'on repère grâce à la date de modification des fichiers)

Structure du Makefile

Un makefile est un ensemble de règles définies par :

```
nom_cible : liste_dependances
<TAB>commandes
```

- `nom_cible` : nom du fichier à générer
- `liste_dependances` : liste des fichiers permettant la génération de la cible
- `commandes` (obligatoirement précédées d'une tabulation) : commande de compilation permettant de générer la cible.

Exemple

Makefile (fichier disponible sur Moodle)

```
matrices : matrices.o matricesio.o operations.o
<tab>gcc matrices.o matricesio.o operations.o -o matrices

matrices.o : matrices.c operations.h matricesio.h commun.h
<tab>gcc -c matrices.c

matricesio.o : matricesio.c matricesio.h commun.h
<tab>gcc -c matricesio.c

operations.o : operations.c operations.h commun.h
<tab>gcc -c operations.c

clean :
<tab>rm -f *.o matrices
```

```
>> make clean
>> make matrices
```

Autres possibilités du makefile

- Définition de variables

```
CC = gcc
LDFLAGS = -lm

EXEC = matrices
OBJ = matrices.o operations.o matricesio.o

$(EXEC) : $(OBJ)
<tab>$(CC) -o $(OBJ) $(EXEC) $(LDFLAGS)
```

- Variables pré-définie :
  - $@ : la cible
  - $< : la première dépendance
  - $^ : toutes les dépendances
- Règles génériques

```
%.o: %.c
<tab>$(CC) $(LDFLAGS) -o $@ -c $<
```

Exemple de makefile plus paramétrable

makefile

```
# options de compilation
CC = gcc
CCFLAGS = -Wall
LIBSDIR =
LDFLAGS = -lm

# fichiers du projet
SRC = matrices.c matricesio.c operations.c
OBJ = $(SRC:.c=.o)
EXEC = matrices

# règle initiale
all: $(EXEC)

# dépendance des .h
matrices.o: operations.h matricesio.h commun.h
matricesio.o: matricesio.h commun.h
operations.o: operations.h commun.h

# règles de compilation
%.o: %.c
	$(CC) $(CCFLAGS) -o $@ -c $<

# règles d'édition de liens
$(EXEC): $(OBJ)
	$(CC) -o $@ $^ $(LIBSDIR) $(LDFLAGS)

# règles supplémentaires
clean:
	rm -f  $(EXEC) *.o
```

Les fichiers

Une entrée/sortie correspond à un tranfert d'information entre la mémoire de la machine et un périphérique (écran, clavier, disque dur, ...)

L'information est traitée sur formes de blocs qui représentent un volume de données tranférées

Un fichier est une suite de blocs stockés sur un disque.

Types de fichiers

- Les fichiers binaires : Ils contiennent l'information brute telle qu'elle est codée dans la mémoire centrale. Ils se présentent comme une suite d'octets mis bout à bout, ils ne sont pas lisibles par l'humain.

- Les fichiers textes : Ils contiennent des données "traduites" pour être lisibles. Ils contiennent des séparateurs (espaces, tabulations, retours à la ligne)

Types d'accès aux fichiers

- Accès séquentiel : On accède aux blocs successivement du premier au dernier. Ainsi pour atteindre une donnée, il faut lire toutes les données précédentes.
- Accès direct : On déplace la position de lecture vers la position voulue avant de lire la donnée.
- Accès indexé : Il existe des index qui "pointent" vers des zones spécifiques du fichier (utilisé pour les gros fichiers, comme les bases de données).

Les structures FILE et FILE *

FILE est une structure contenant plusieurs champs nécessaires aux entrées/sorties :

- Un pointeur sur la mémoire tampon associée au fichier
- Un pointeur sur la position courante dans le fichier
- Un mode d'accès au fichier (lecture, écriture, ...)
- Un indicateur d'erreur et un indicateur de fin de fichier

Les fonctions C ne manipulent que des pointeurs sur FILE : FILE *

```
int main() {
  FILE *fid;
```

Entrées/Sorties standards

3 variables d'entrées-sorties de type `FILE *` sont prédéfinies dans le langage C (bibliothèque `stdio.h`) et sont toujours utilisables :

- `stdin` : fichier d'entrée standard, c'est à dire le clavier.
- `stdout` : fichier de sortie standard, c'est à dire l'écran.
- `stderr` : fichier d'erreurs, par défaut, l'écran.

Toutes les opérations faites sur les fichiers peuvent être faites sur les entrées-sorties prédéfinies.

Liste des principales fonctions

```
int n,p,fend;
FILE *fid;
```

nom	description	exemple
`fopen`	ouverture d'un fichier	`fid = fopen("fic.txt","r");`
`fclose`	fermeture d'un fichier	`fclose(fid);`
`fwrite`	écriture dans un fichier binaire	`fwrite(&n,sizeof(int),1,fid);`
`fprintf`	écriture dans un fichier texte	`fprintf(fid,"%d",n);`
`fread`	lecture dans un fichier binaire	`fread(&p,sizeof(int),1,fid);`
`fscanf`	lecture dans un fichier texte	`fscanf(fid,"%d",&p);`
`feof`	test de fin de fichier	`fend = foef(fid);`

Ces fonctions appartiennent à la bibliothèque `stdio.h`

Ouverture/Fermeture

- Ouverture avec `fopen()` :
  `FILE * fopen(char nomFichier[], char modeOuverture[]);`
  → `fopen()` retourne `NULL` si l'opération échoue.
- Fermeture avec `fclose()` :
  `fclose (FILE * fid);`
  → `fclose()` retourne 0 si tout s'est bien déroulé et `EOF` sinon.

```
int main() {
 char nom[]="fic.txt";
 FILE *fid;
 fid=fopen(nom,"r");
 fclose(fid);
}
```

Modes d'ouverture

```
//Ouverture en mode lecture
fid=fopen(nom,"r");
```

- **"r"** : ouverture du fichier en lecture.
- **"w"** : ouverture du fichier en écriture
  S'il n'existe pas, alors création, sinon, destruction.
- **"a"** : ouverture en ajout (écriture à la fin du fichier)
  S'il n'existe pas, alors création, sinon positionnement à la fin.
- "r+" : ouverture en lecture et écriture
  Positionnement au début du fichier.
- "w+" : ouverture en lecture et écriture
  S'il n'existe pas, alors création, sinon, destruction.
- "a+" : Ouverture en lecture et ajout
  S'il n'existe pas, alors création. La lecture est positionnée au début, l'écriture à la fin.

Lecture/Ecriture dans un fichier binaire

- Lecture avec `fread()` :
  `int fread (type *adr, int taille, int nbloc, FILE *fid);`
  → Place les données lues dans `adr`.
  → retourne le nombre de valeurs lues.
- Ecriture avec `fwrite()` :
  `int fwrite (type *adr, int taille, int nbloc, FILE *fid);`
  → Ecrit les données de `adr` dans le fichier.
  → retourne le nombre de blocs écrits.
- `taille` : taille du bloc à écrire ou à lire (généralement déterminé par `sizeof()`).
- `nbloc` : nombre de blocs à écrire ou à lire (ex : dimension d'un tableau).

Lecture/Ecriture formatée dans un fichier texte

- Lecture avec `fscanf()` (comme `scanf`) :
  `int fscanf (FILE *fid, format, ...);`
- Ecriture avec `fprintf()` (commet `printf`) :
  `int fprintf (FILE *fid, format, ...);`
- `format` : Même utilisation que dans scanf ou printf
- Différence avec le binaire : les fichiers sont lisibles par l'humain.