Programmation langage C

Section 13 : Programmation modulaire

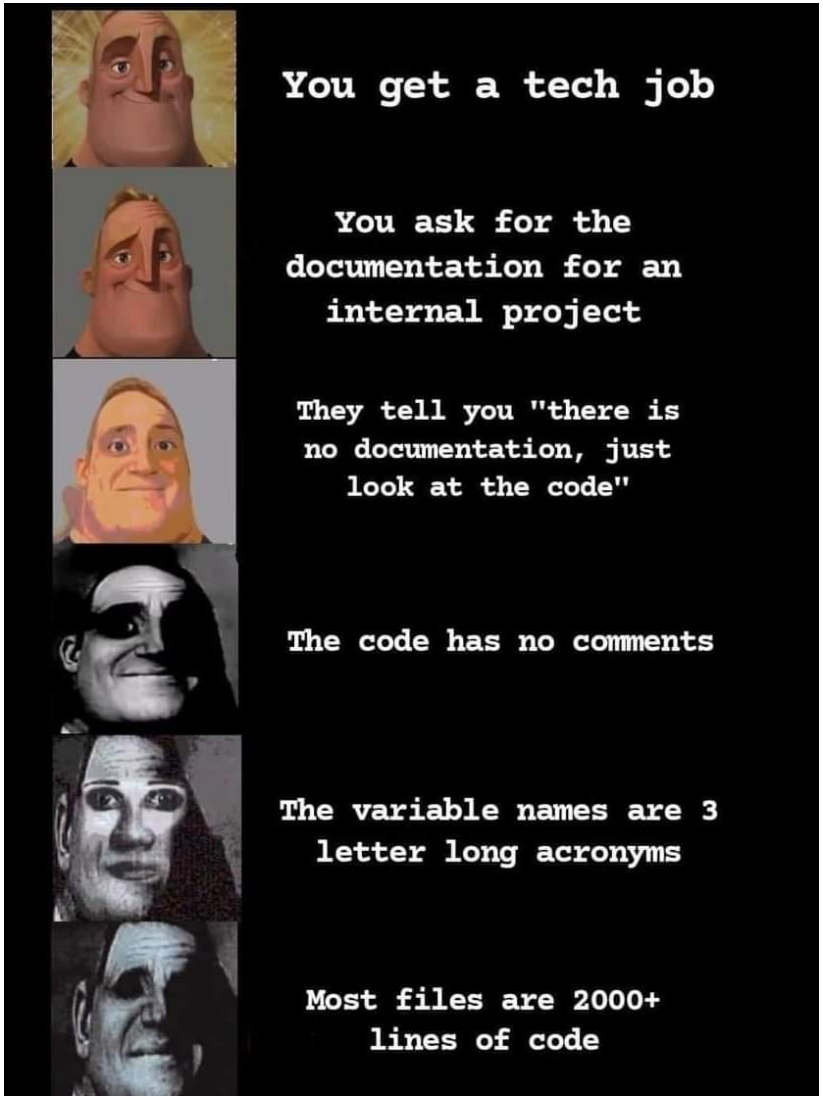
Présentation de **Kevin TRANCHO**

dispensé en classe de seconde année

à l'**ESGI** Paris

(Année scolaire 2022 - 2023)

Introduction



Introduction

Comment ne plus tout coder dans un fichier et séparer proprement le code ?

Code à séparer

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

void maFonction() {
        printf("Ma Fonction\n");
}

int main() {
  maFonction();
  exit(EXIT_SUCCESS);
}
```

Séparation en deux fichiers sources ⋆[1]

```c
/* main.c */

#include <stdlib.h>

extern void maFonction();

int main() {
  maFonction();
  exit(EXIT_SUCCESS);
}
```

```c
/* mes_fonctions.c */

#include <stdio.h>

void maFonction() {
  printf("Ma Fonction\n");
}
```

```
gcc -o executable main.c mes_fonctions.c
```

Séparation en deux fichiers sources ⋆[1]

```c
/* main.c */

#include <stdlib.h>

extern void maFonction();

int main() {
  maFonction();
  exit(EXIT_SUCCESS);
}
```

```c
/* mes_fonctions.c */

#include <stdio.h>

void maFonction() {
  printf("Ma Fonction\n");
}
```

```
gcc -o executable main.c mes_fonctions.c
```

Séparation en deux fichiers sources ⋆[1]

```c
/* main.c */

#include <stdlib.h>

extern void maFonction();

int main() {
  maFonction();
  exit(EXIT_SUCCESS);
}
```

```c
/* mes_fonctions.c */

#include <stdio.h>

void maFonction() {
  printf("Ma Fonction\n");
}
```

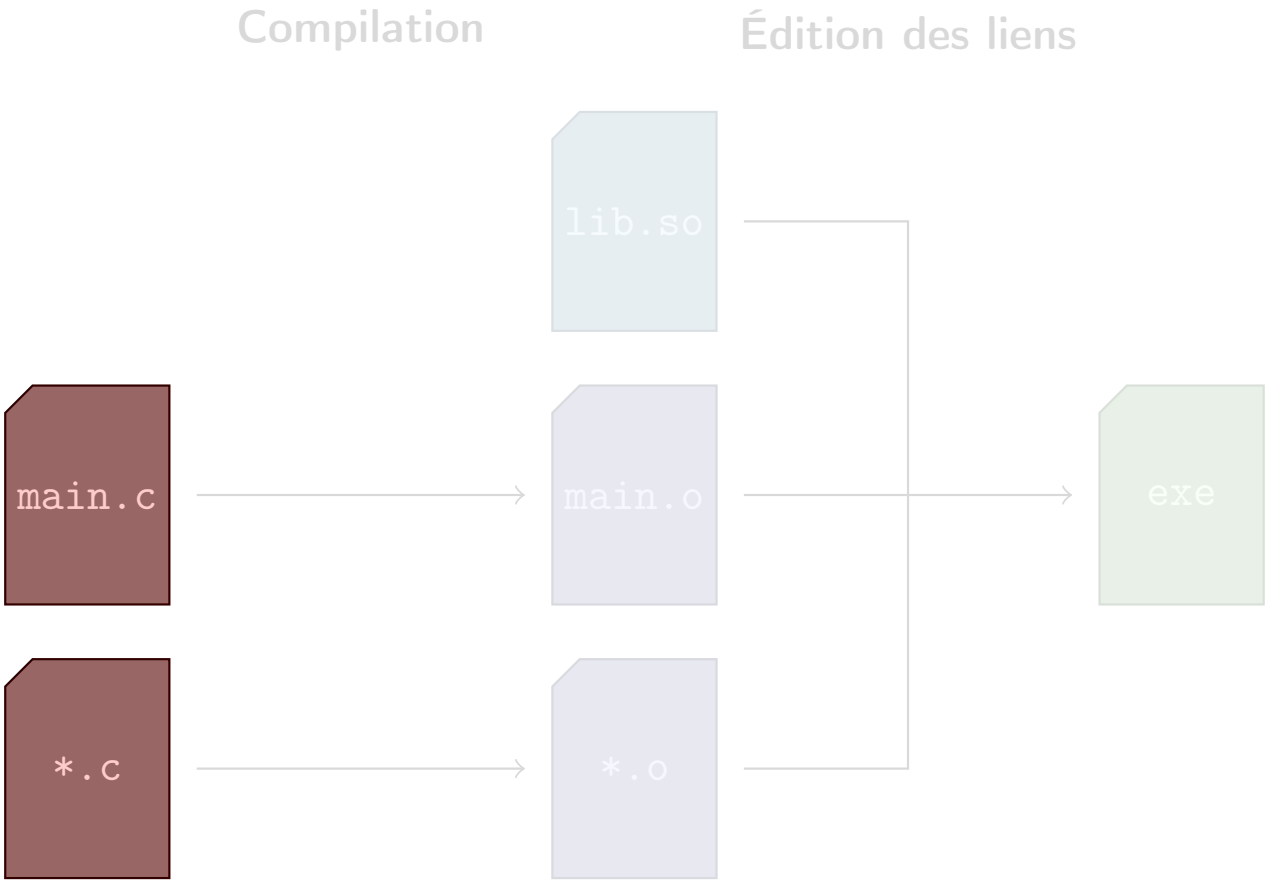
```
gcc -o executable main.c mes_fonctions.c
```

extern : importation variable globale externe

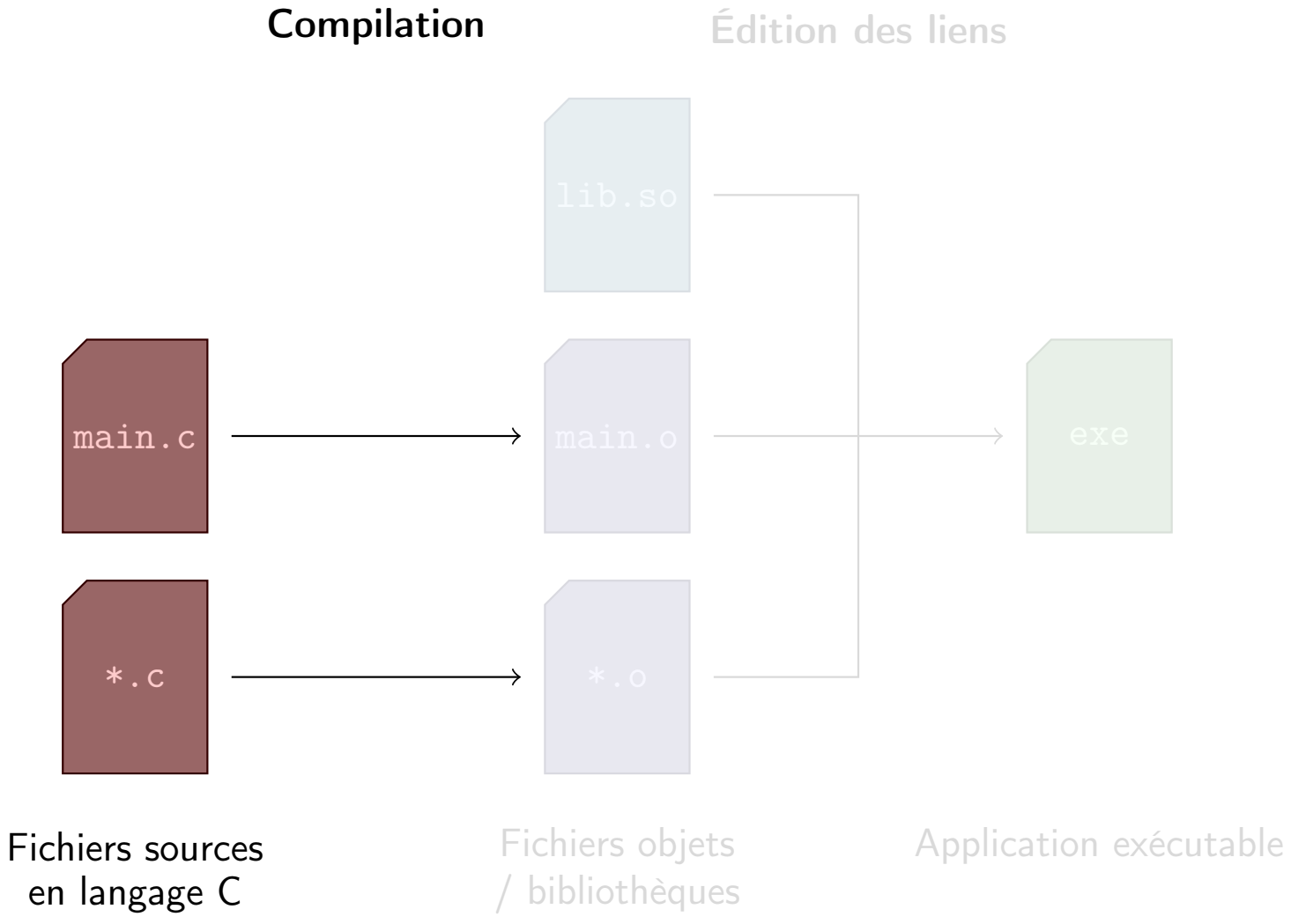
```c
/* main.c */

#include <stdlib.h>

extern int variable;

extern void
↪  maFonction(int);

int main() {
  maFonction(variable);
  exit(EXIT_SUCCESS);
}
```

```c
/* mes_fonctions.c */

#include <stdio.h>

int variable = 42;

void maFonction(int v) {
  printf("Ma Fonction
  ↪  %d\n", v);
}
```

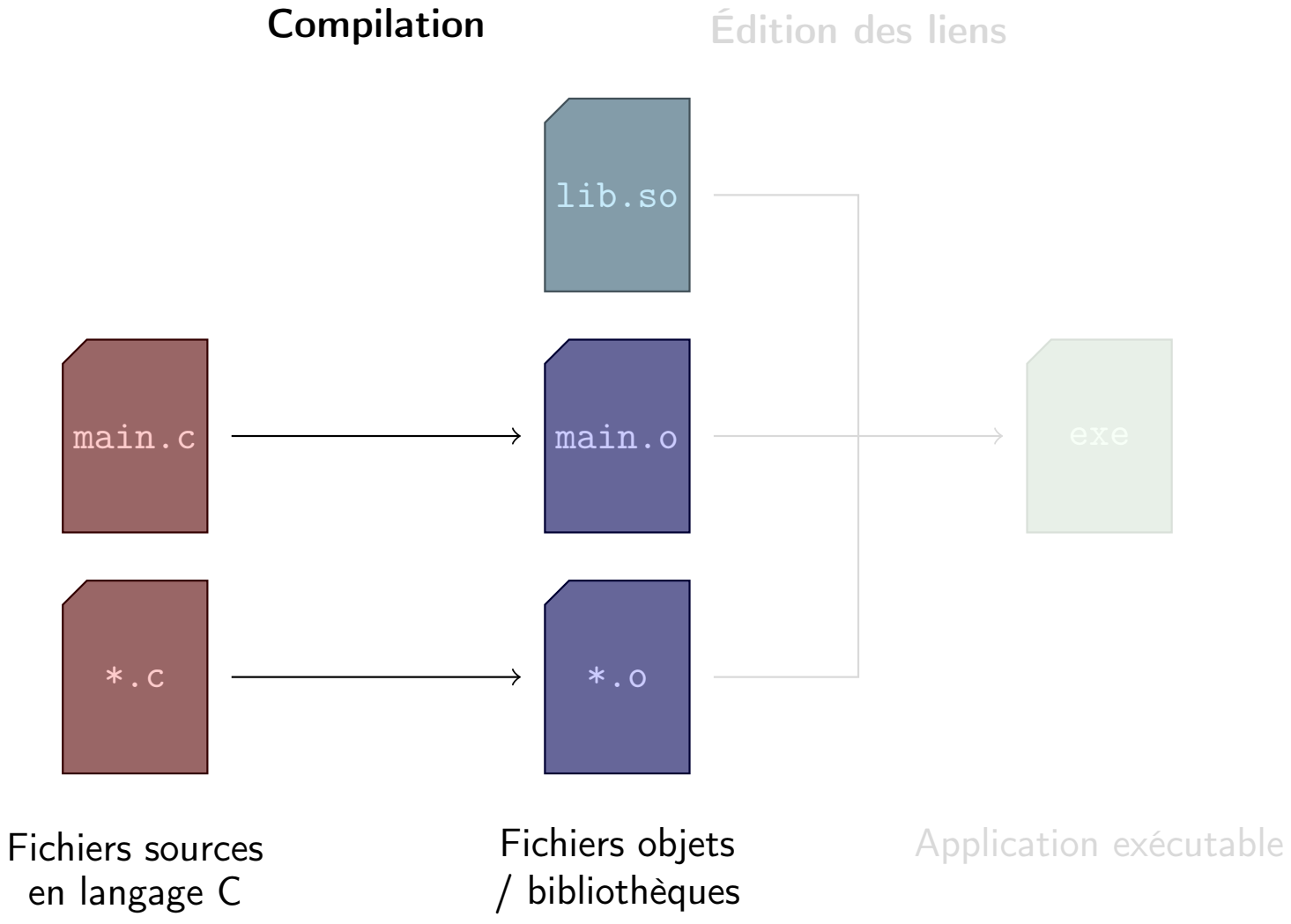
Rappel : Compilation et linkage



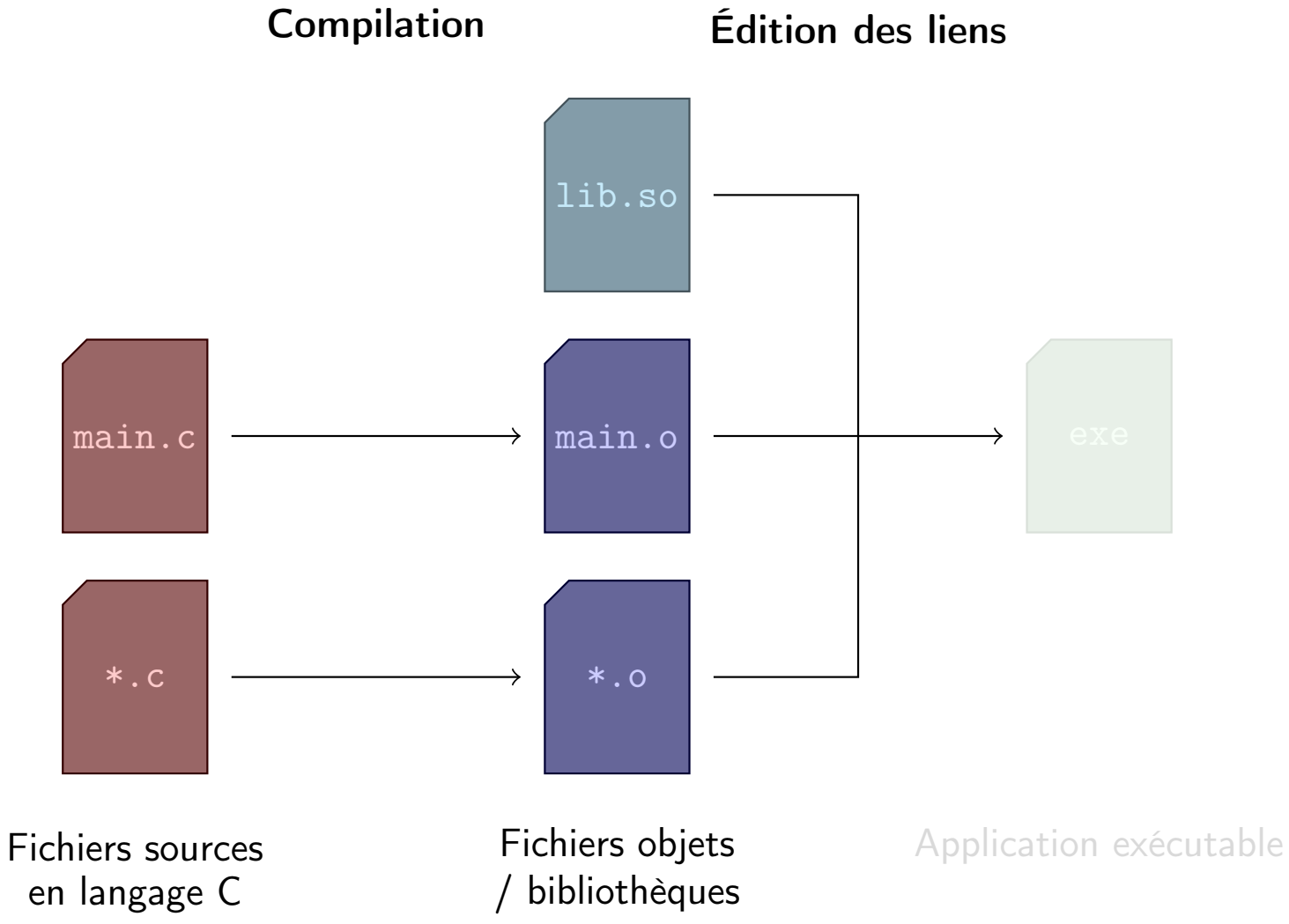
Rappel : Compilation et linkage



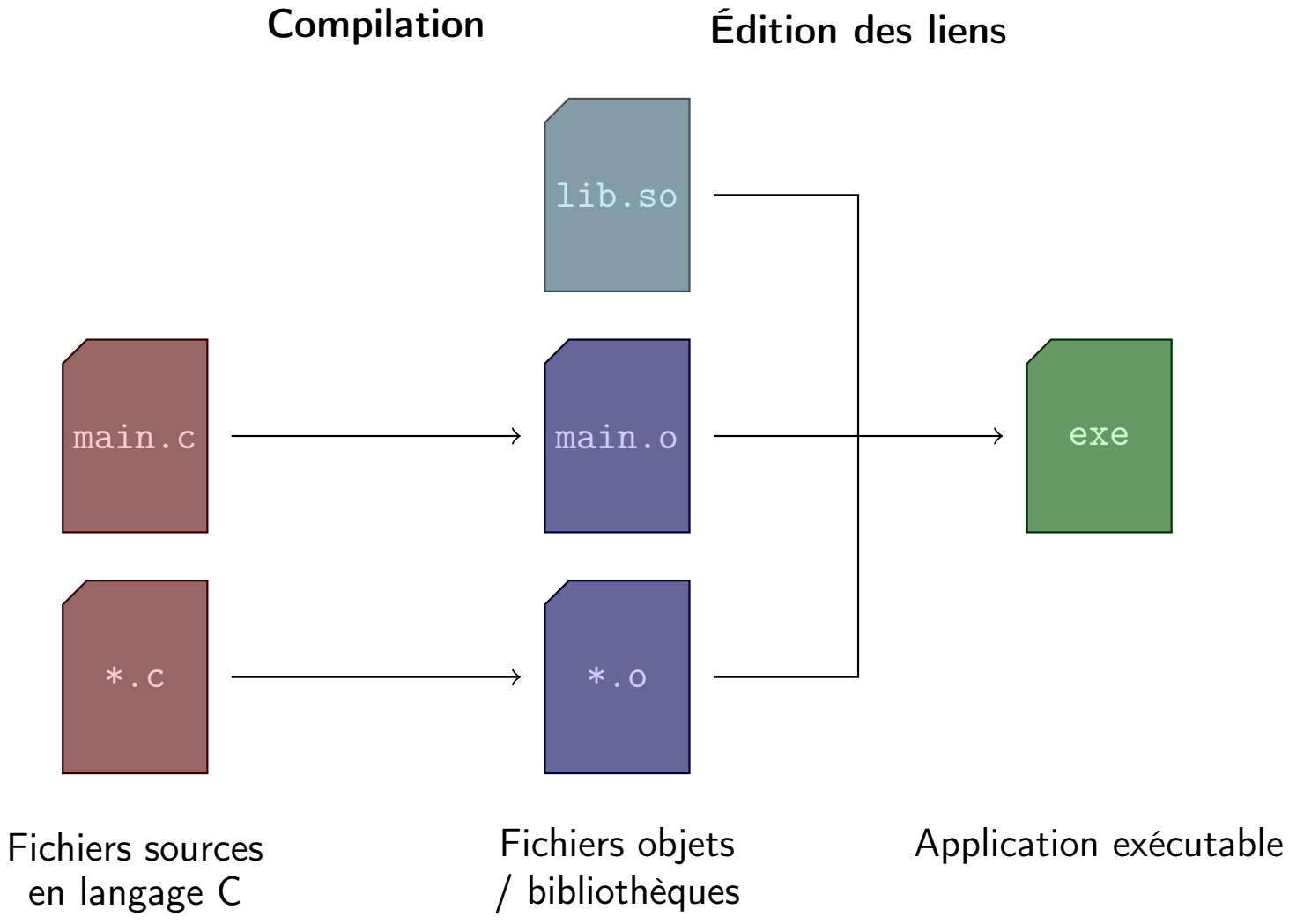
Rappel : Compilation et linkage



Rappel : Compilation et linkage



Rappel : Compilation et linkage



Compilation avec problème d'édition des liens

```
# gcc -o executable main.c
/tmp/cc5vxQgu.o : Dans la fonction « main » :
main.c:(.text+0xa) : référence indéfinie vers «
 ↪  maFonction »
collect2: error: ld returned 1 exit status
```

Compilation séparée $\star^2$

```
# gcc -c mes_fonctions.c
# gcc -c main.c
# gcc -o executable main.o mes_fonctions.o
```

Visualiser table des symboles $\star^2$

```
# readelf -s main.o

La table de symboles « .symtab » contient 13 entrées :
   Num:    Valeur          Tail Type    Lien   Vis      Ndx Nom
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
...
     8: 0000000000000000    27 FUNC    GLOBAL DEFAULT    1 main
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND variable
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND
    ↪  _GLOBAL_OFFSET_TABLE_
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND maFonction
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit
```

Importer une bibliothèque : stdio

```c
#include <stdio.h>

int main() {
  printf("Hello ESGI !\n");
  return 0;
}
```

Importer une bibliothèque : stdio $\star^3$

6 lignes, non ?

```
gcc -o main.i -E -P main.c
```

Importer une bibliothèque : la vérité

```c
/* main.i */
typedef long unsigned int size_t;
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
... /* environ 200 lignes */

int main() {
 printf("Hello ESGI !\n");
 return 0;
}
```

Importer une bibliothèque : pré-traitement

Pré-traitement des directives préprocesseur.

Les directives préprocesseur sont des instructions générant du code en langage C avant la compilation.

C'est ensuite ce code qui sera compilé.

Importation fonctionnelle de fonctions sans directives préprocesseur

```
extern int printf(const char *, ...);

int main() {
        printf("Hello ESGI !\n");
        return 0;
}
```

Directive préprocesseur include

La directive préprocesseur `include` permet de recopier le contenu d'un fichier à l'emplacement du fichier C courant où elle est appelée. À utiliser avec parcimonie, en général pour inclure l'entête de modules :

- `#include <entete.h>` : permet l'inclusion d'une bibliothèque présente ou installée sur la machine.
- `#include "entete.h"` : est réservé à l'inclusion relative de vos propres fichiers d'entête.

inclusion de mes fonctions

```c
/* main.c */

#include <stdlib.h>
#include "mes_fonctions.h"

int main() {
  maFonction(variable);
  exit(EXIT_SUCCESS);
}
```

```c
/* mes_fonctions.h */

extern int variable;

extern void maFonction(int);
```

```c
/* mes_fonctions.c */

#include <stdio.h>

int variable = 42;

void maFonction(int v) {
  printf("Ma Fonction %d\n", v);
}
```

inclusion de mes fonctions

```c
/* main.c */

#include <stdlib.h>
#include "mes_fonctions.h"

int main() {
  maFonction(variable);
  exit(EXIT_SUCCESS);
}
```

```c
/* mes_fonctions.h */

extern int variable;

extern void maFonction(int);
```

```c
/* mes_fonctions.c */

#include <stdio.h>

int variable = 42;

void maFonction(int v) {
  printf("Ma Fonction %d\n", v);
}
```

inclusion de mes fonctions

```c
/* main.c */

#include <stdlib.h>
#include "mes_fonctions.h"

int main() {
  maFonction(variable);
  exit(EXIT_SUCCESS);
}
```

```c
/* mes_fonctions.h */

extern int variable;

extern void maFonction(int);
```

```c
/* mes_fonctions.c */

#include <stdio.h>

int variable = 42;

void maFonction(int v) {
  printf("Ma Fonction %d\n", v);
}
```

Directive préprocesseur define

```c
#include <stdio.h>
#include <stdlib.h>
#define TAILLE 10

int main() {
  int i;
  for(i = 0; i < TAILLE; ++i) {
    printf("%d\n", i);
  }
  exit(EXIT_SUCCESS);
}
```

Directive préprocesseur undef

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

int main() {
#define NOMBRE 42
  printf("define NOMBRE %d\n", NOMBRE);
#undef NOMBRE
#define NOMBRE 13.37
  printf("define NOMBRE %g\n", NOMBRE);
#undef NOMBRE
#define NOMBRE "1234"
  printf("define NOMBRE %s ?\n", NOMBRE);
  exit(EXIT_SUCCESS);
}
```

Directive préprocesseur define : synonyme fonction

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

#define print puts

int main() {
  print("Hello ESGI");
  /* Python de langage C ! */
  exit(EXIT_SUCCESS);
}
```

Directive préprocesseur define : morceau de code

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

#define INSTRUCTIONS printf("Hello ESGI\n"); \
                     exit(EXIT_SUCCESS);

int main() {
  INSTRUCTIONS
}
```

Directive préprocesseur define : macros

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

#define min(a, b) (a < b) ? a : b

int main() {
  int a = 42, b = 1337;
  printf("min(%d, %d) = %d\n", a, b, min(a, b));
  float c = 13.37, d = 42.;
  printf("min(%g, %g) = %g\n", c, d, min(c, d));
  exit(EXIT_SUCCESS);
}
```

Directive préprocesseur define : macros - précautions $\star^4$

```c
int fonction_min(int a, int b) {
  return (a < b) ? a : b;
}

#define macro_min(a, b) ((a) < (b)) ? (a) : (b)

int main() {
  int v, a, b;
  v = fonction_min(a = getchar() - '0', b = getchar() - '0');
  printf("fonction_min(%d, %d) = %d\n", a, b, v);
  v = macro_min(a = getchar() - '0', b = getchar() - '0');
  printf("macro_min(%d, %d) = %d\n", a, b, v);
  exit(EXIT_SUCCESS);
}
```

Directive préprocesseur define : macros - précautions

```
24246
fonction_min(4, 2) = 2
macro_min(6, 4) = 6
```

Directive préprocesseur define : macros - précautions

```
int fonction_min(int a, int b) {
  return (a < b) ? a : b;
}

int main() {
  int v, a, b;
  v = fonction_min(a = getchar() - '0', b = getchar() - '0');
  printf("fonction_min(%d, %d) = %d\n", a, b, v);
  v = ((a = getchar() - '0') < (b = getchar() - '0')) ? (a =
  ↪  getchar() - '0') : (b = getchar() - '0');
  printf("macro_min(%d, %d) = %d\n", a, b, v);
  exit(0);
}
```

Directive préprocesseur define : code en texte

```c
#define FAIRE_CALCUL(exp) printf("%s = %d\n", #exp, exp)

int main() {
  int valeur = 4;
  FAIRE_CALCUL(valeur * valeur + 2 * valeur + 1);
  exit(EXIT_SUCCESS);
}
```

```
valeur * valeur + 2 * valeur + 1 = 25
```

Directive préprocesseur define : concaténer code

```c
#define macro_abs(a) (((a) < 0) ? -(a) : (a))
#define macro_carre(a) ((a) * (a))

#define call(f, x) printf("%s(%d) = %d\n", #f, x, macro_##f(x))

int main() {
  call(abs, -5);
  call(carre, -5);
  exit(EXIT_SUCCESS);
}
```

```
abs(-5) = 5
carre(-5) = 25
```

Directive préprocesseur conditionnelle $\star^5$

```
#define DEBUG
int main() {
#if defined VERBOSE
  fprintf(stderr, "Entrée dans le main\n");
#endif
#if defined VERBOSE
  printf("42, la réponse à la vie !\n");
#elif defined DEBUG
  printf("42, vous savez pourquoi.\n");
#else
  printf("42 !\n");
#endif
#if defined VERBOSE
  fprintf(stderr, "Sortie du main\n");
#endif
  exit(EXIT_SUCCESS);
```

Directive préprocesseur conditionnelle

```
int main() {
  printf("42, vous savez pourquoi.\n");
  exit(0);
}
```

Ajout d'une directive préprocesseur à la compilation

```
gcc -o main.i -E -P main.c -DVERBOSE
```

```c
int main() {
  fprintf(stderr, "Entrée dans le main\n");
  printf("42, la réponse à la vie !\n");
  fprintf(stderr, "Sortie du main\n");
  exit(0);
}
```

Abréviation directive préprocesseur conditionnelle

```
#if defined NOM
#ifdef NOM

#if !defined NOM
#ifndef NOM
```

Créer un module

- Fournir un fichier d'entête `.h` pour utiliser les fonctionnalités du module.
- Implémenter les fonctionnalités dans un fichier `.c` du même nom.

Créer un module : fichier d'entête .h

```c
#ifndef DEF_HEADER_MODULE
#define DEF_HEADER_MODULE
/* Protection du module */
/**
  Documentation du module et
↪  auteurs
*/
/* Macros publiques */
#define abs(x) (((x) < 0) ? -(x)
↪  : (x))

/* Types publiques du module */
typedef struct Point Point;
struct Point {
  float x;
  float y;
};

/* Variables publiques du module
↪  */
extern Point origine;

/* Fonctionnalités publiques du
↪  module */

/* Affiche un point dans la
↪  sortie standard */
extern void Point_afficher(const
↪  Point * point);

#endif
```

Créer un module : implémentation .c

```c
#include "module.h"
/* Inclusion des déclarations du
↪  module */

#include <stdio.h>
/* Autres inclusions */

/* Définition des variables
↪  globales relatives au module
↪  */
Point origine = {0, 0};

/* Fonctionnalité privée au
↪  module par le mot-clé static
↪  */
static void Point_print(FILE *
↪  flow, const Point * point) {
  if(! point) {
    fprintf(flow, "(nil)");
  }
  fprintf(flow, "(%g, %g)",
  ↪  point->x, point->y);
}

/* Définition des fonctionnalités
↪  annoncées par l'entête */
void Point_afficher(const Point *
↪  point) {
  Point_print(stdout, point);
}
```

Makefiles

Makefile

```
executable :
        gcc -o executable *.c
```

```
make
```

Makefiles

Makefile

```
executable :
	gcc -o executable *.c
```

```
make
```

Makefiles

Un Makefile : liste de cibles, dépendances associées et commandes à exécuter. Chaque commande est précédée d'une **tabulation**.

```
cible: dépendances
	commande
	...
	commande
```

Exemple de Makefile pour projet C

```
executable : main.o module.o
	gcc -o executable main.o module.o

main.o : main.c module.h
	gcc -c main.c

module.o : module.c module.h
	gcc -c module.c

clean :
	rm -rf *.o
```

Symboles dans le Makefile

Symbole	Correspondance
`$@`	Cible
`$^`	Toutes les dépendances
`$<`	Première dépendance
`%`	Règle générique

Exemple de Makefile pour projet C avec symboles

```
executable : main.o module.o
	gcc -o $@ $^

main.o : main.c module.h
	gcc -c $<

module.o : module.c module.h
	gcc -c $<
```

Exemple de Makefile pour projet C avec symboles et règle générique

```
executable : main.o module.o
	gcc -o $@ $^

main.o : main.c module.h
module.o : module.c module.h

%.o : %.c
	gcc -c $<
```

Variables dans un Makefile

- définition `VARIABLE=VALEUR`
- appel `$(VARIABLE)`

Pertinentes dans notre cas :

- le compilateur : `CC=gcc` (ou par exemple `CC=clang`).
- les drapeaux de compilation et optimisations dans `CFLAGS` (optimisations : `-O1`, `-O2` et éventuellement `-O3`).
- les bibliothèques dans `CLIBS`.
- le nom de l'exécutable.

Makefile pertinent

```
CC= gcc
CFLAGS= -O2 -Wall -Wextra -Werror -ansi
CLIBS= -lm
EXE= executable

$(EXE) : main.o module.o
	$(CC) $(CFLAGS) -o $@ $^ $(CLIBS)

main.o : main.c module.h
module.o : module.c module.h

%.o : %.c
	$(CC) $(CFLAGS) -c $<
```

Makefile pertinent avec dossiers

```
CC= gcc
CFLAGS= -O2 -Wall -Wextra -Werror -ansi
CLIBS= -lm
EXE= executable
OBJ= obj/
SRC= src/
INCL= include/

$(EXE) : $(OBJ)main.o $(OBJ)module.o
	$(CC) $(CFLAGS) -o $@ $^ $(CLIBS)

$(OBJ)main.o : $(SRC)main.c $(INCL)module.h
$(OBJ)module.o : $(SRC)module.c $(INCL)module.h

$(OBJ)%.o : $(SRC)%.c
	$(CC) $(CFLAGS) -o $@ -c $<

clean :
	rm -rf $(OBJ)*.o
	rm -rf $(EXE)
```

Makefile magique

```
CC= gcc
CFLAGS= -O2 -Wall -Wextra -Werror -ansi
CLIBS= -lm
EXE= executable
OBJ= obj/
SRC= src/
INCL= include/
FILEC:= $(wildcard $(SRC)*.c)
FILEH:= $(wildcard $(INCL)*.h)
FILEO:= $(patsubst $(SRC)%.c,$(OBJ)%.o,$(FILEC))

$(EXE) : $(FILEO)
	$(CC) $(CFLAGS) -o $@ $^ $(CLIBS)

$(OBJ)main.o : $(SRC)main.c $(FILEH)
	$(CC) $(CFLAGS) -o $@ -c $<

$(OBJ)%.o : $(SRC)%.c $(INCL)%.h
	$(CC) $(CFLAGS) -o $@ -c $<

clean :
	rm -rf $(OBJ)*.o
	rm -rf $(EXE)
```

Questions

Avez-vous des questions ?

Exercices

- Travailler sur les exercices sur la programmation modulaire (section 13) du support de cours.
- Si les exercices de la section 13 sont terminés :
  - Avancer sur les sections 14 et 15.
  - Si cours terminé : Avancer sur le projet.
  - Si projet terminé avec certitude de 21 / 20 : le pousser plus loin.

Annexe


EVERY GROUP PROJECT
DOES 99% OF THE WORK
HAS NO IDEA WHAT'S GOING ON THE WHOLE TIME
SAYS HE'S GOING TO HELP BUT HE'S NOT
DISAPPEAR AT THE VERY BEGINNING AND DOESN'T SHOW UP AGAIN TIL THE VERY END