MAKE PAR L'EXEMPLE

FRÉDÉRIC DROUHIN

Merci à Michel Hassenforder pour la relecture

SOMMAIRE

SOMMAIRE

PROCESSUS DE COMPILATION

FRÉDÉRIC DROUHIN

Différentes Phases

Pré-Processing

Compilation proprement dite

Assemblage

Edition des liens

Notions de libraires (par exemple GTK)

PRÉ-PROCESSING

game.c (v1)

```
#include <stdio.h>

int main() {
    printf("Hello World !");
    return 0;
}
```

game.c (v2)

```
#include <stdio.h>

#define carre(X) X*X
#define MAXVAL 100

int main() {
    int n = 5;
    printf("%d\n", carre(n+3) );
}
```

Directrive → #include

- Permet d'inclure un fichier pour ajouter le prototype des fonctions
- Par exemple : int printf(const char* format, ...);
- Attention le code n'est pas ajouté ici !

Directive → #define

- Le pré processeur va rechercher la première chaine de caractères dans le code source et la remplacer littéralement par la seconde.

Directives #...

- S'adresse au pré-processing

La Compilation proprement Dite

- **Terme générique**
  - Création d'un exécutable
  - Abus de langage
- **Compilation est l'une des 4 étapes :**
  - Pré-Processing
  - **Compilation**
  - Assemblage
  - Edition des liens
- **Permet de produire des instructions pour le processeur**

```
        .file   "game.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello World !"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

L'ASSEMBLAGE

- **Langage machine**
- **Exécuté par le microprocesseur**

$> gcc -c game.c -o game.o

game.o est un fichier objet

```
7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00
01 00 3E 00 01 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 10 03 00 00 00 00 00 00
00 00 00 00 40 00 00 00 00 00 40 00 0E 00 0D 00
F3 0F 1E FA 55 48 89 E5 48 8D 3D 00 00 00 00 E8
00 00 00 00 B8 00 00 00 00 5D C3 48 65 6C 6C 6F
20 57 6F 72 6C 64 20 21 00 00 47 43 43 3A 20 28
55 62 75 6E 74 75 20 39 2E 33 2E 30 2D 31 30 75
62 75 6E 74 75 32 29 20 39 2E 33 2E 30 00 00 00
04 00 00 00 10 00 00 00 05 00 00 00 47 4E 55 00
02 00 00 C0 04 00 00 00 03 00 00 00 00 00 00 00
14 00 00 00 00 00 00 00 01 7A 52 00 01 78 10 01
1B 0C 07 08 90 01 00 00 1C 00 00 00 1C 00 00 00
00 00 00 00 1B 00 00 00 00 45 0E 10 86 02 43 0D
06 52 0C 07 08 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 00 00 00 04 00 F1 FF 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 03 00 01 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 03 00 03 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 03 00 04 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 03 00 05 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 03 00 07 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 03 00 08 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 03 00 09 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

L'Édition de Lien

- **Produit l'exécutable**
- **Ajout de librairies externes**
  - Par exemple (de manière implicite) la fonction printf

```
$> gcc -c game.c -o game
```

  - Par exemple de manière explicite, les fonctions de libpthread

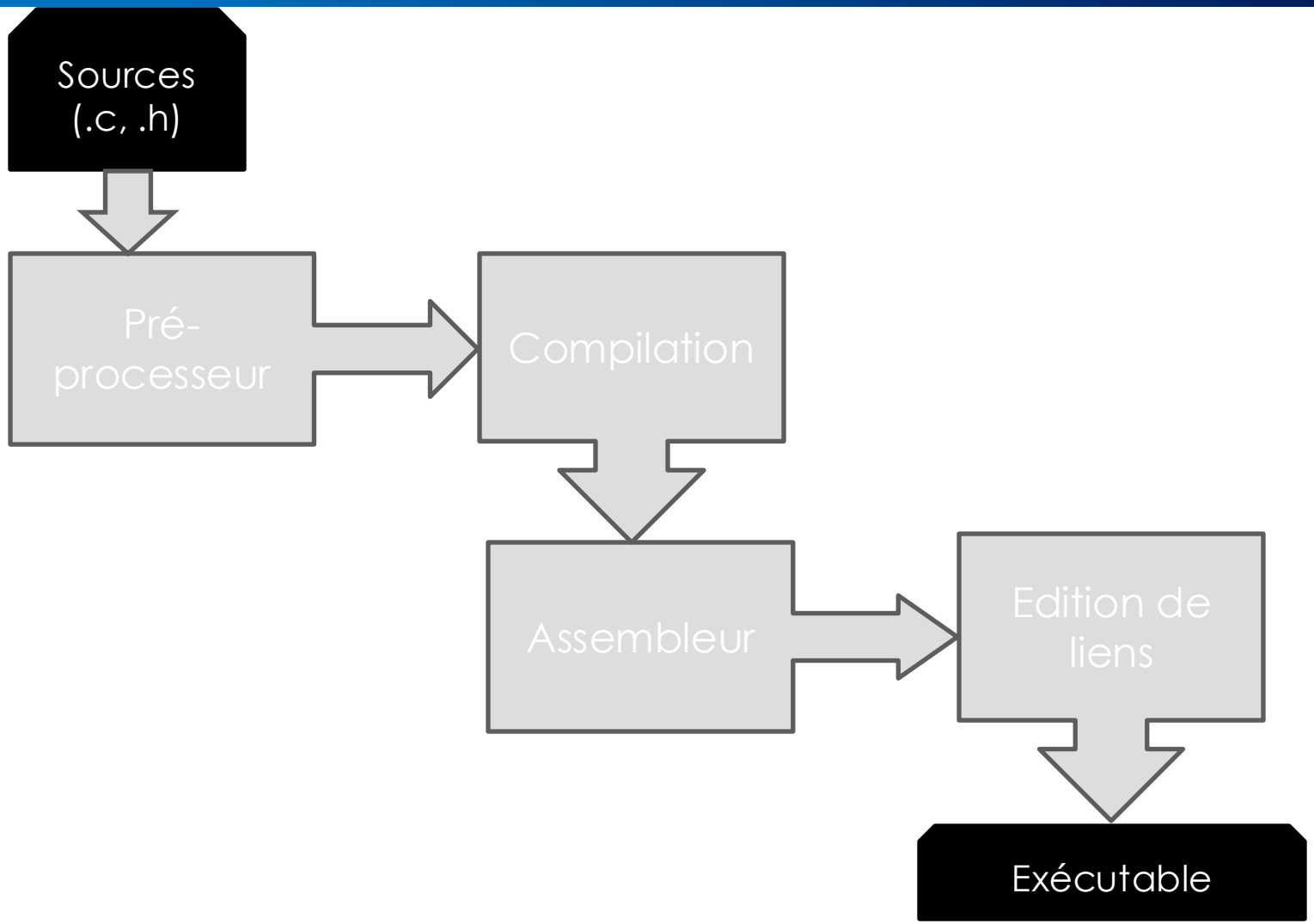
```
$> gcc -c game.c -o game -lpthread
```

Les Libraires

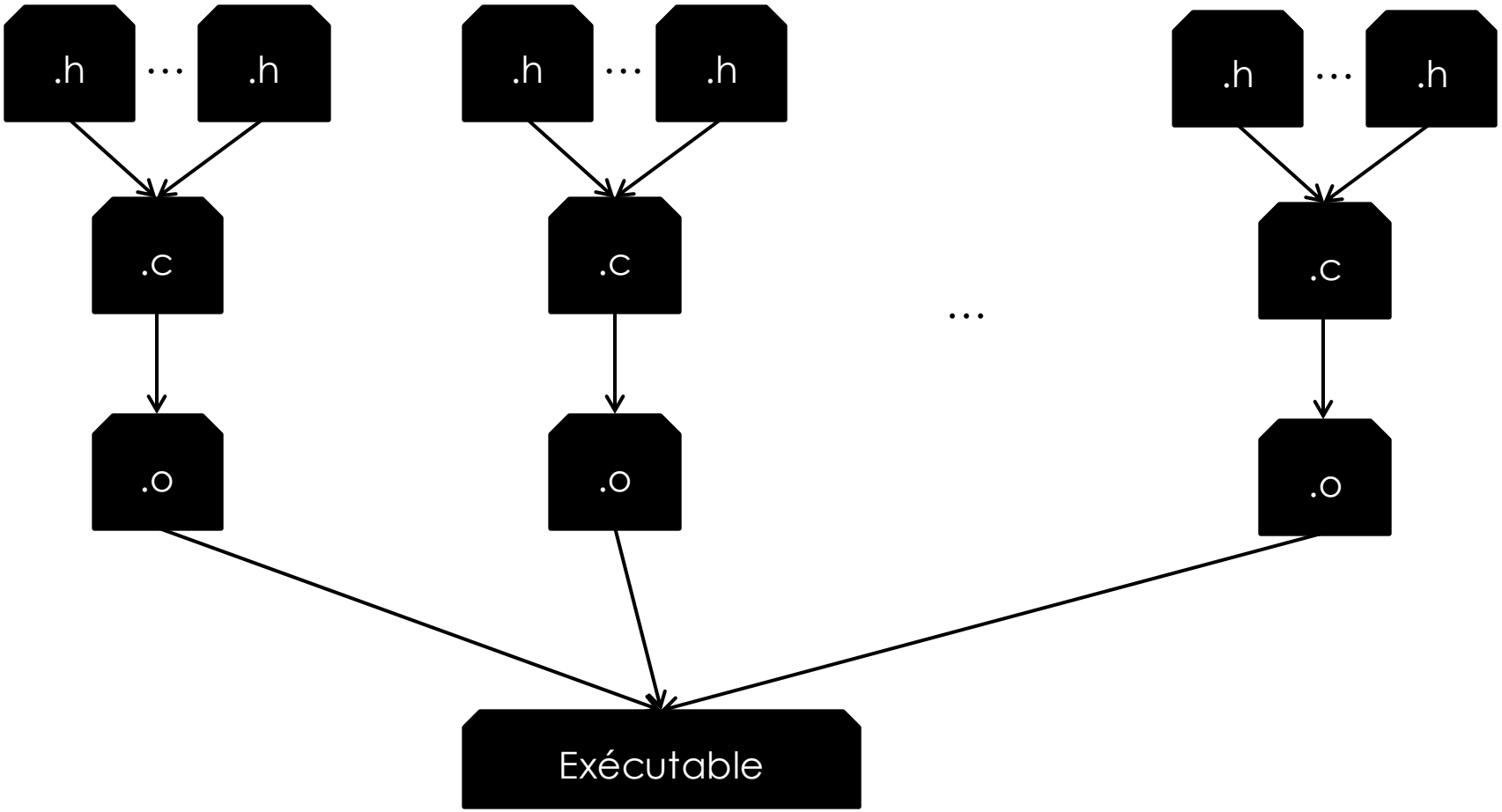
- **.a = statique (GCC Linux)**
- **.so = dynamique (GCC Linux)**
- **.lib = statique (Microsoft Windows)**
- **.dll = dynamique (Microsoft Windows)**

- **.so, .dll : librairie dynamique**
  - Chargez en mémoire au moment de l'exécution
- **.o, .lib : librairie statique**
  - Copiez dans l'exécutable finale

En Résumé

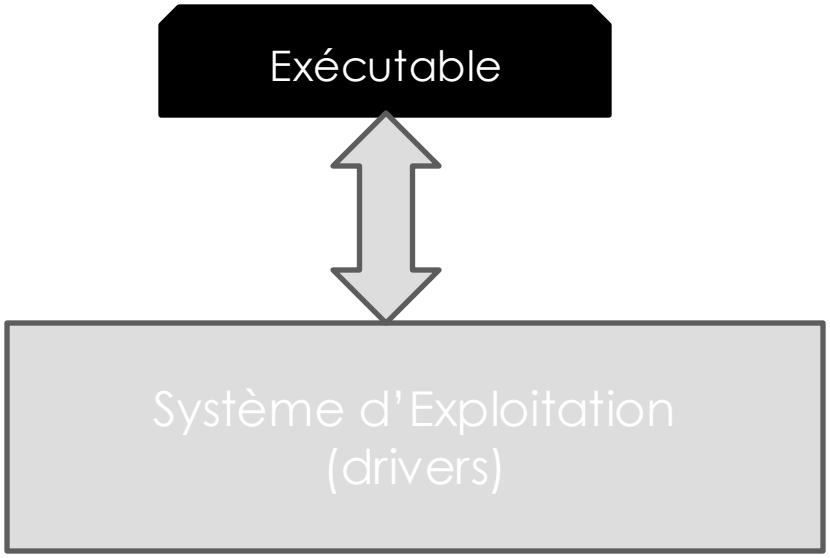

Sources
(.c, .h)
Pré-
processeur
Compilation
Assembleur
Edition de
liens
Exécutable

En Résumé



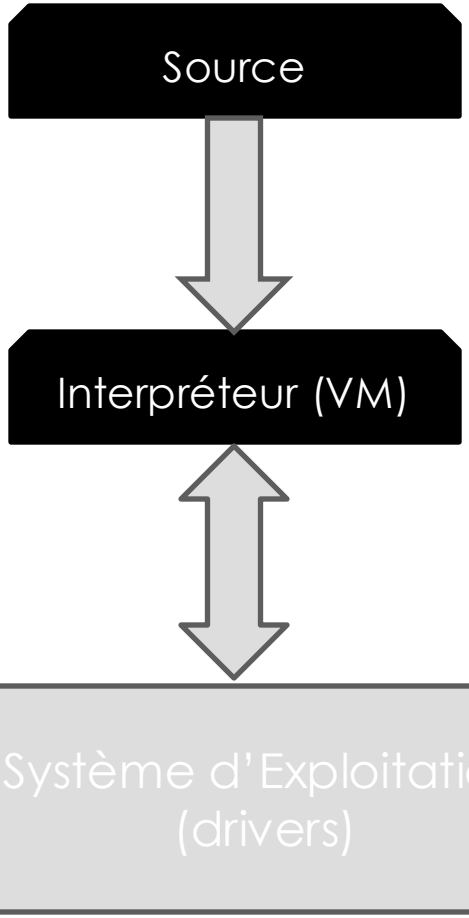
L'Exécution

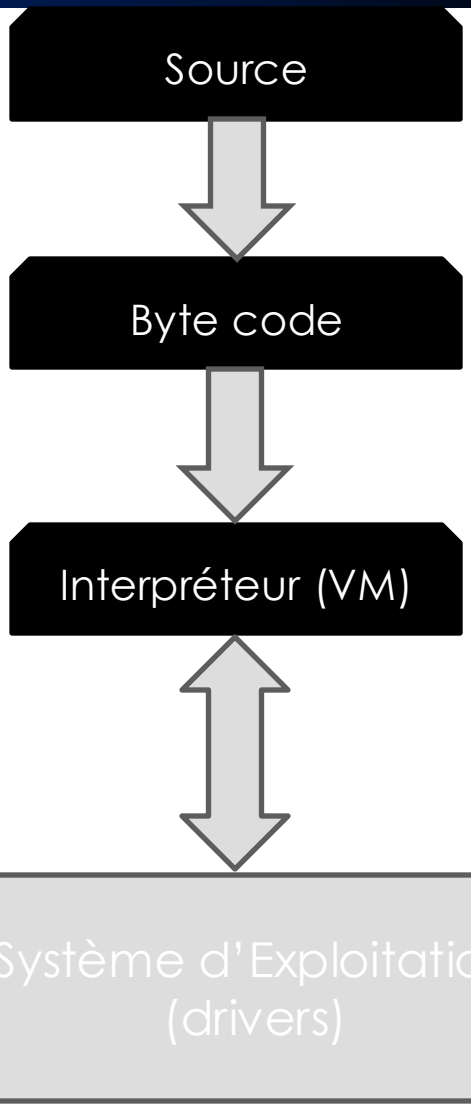
C
C++
Rust
Go
Fortran



Et l'Interpréteur

~~Python~~
Javascript
Bash
PHP
Perl





Java
C#
Python

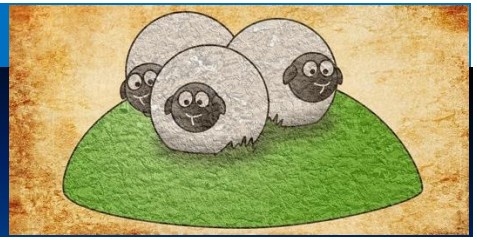
SOMMAIRE

Un rapide point sur la compilation

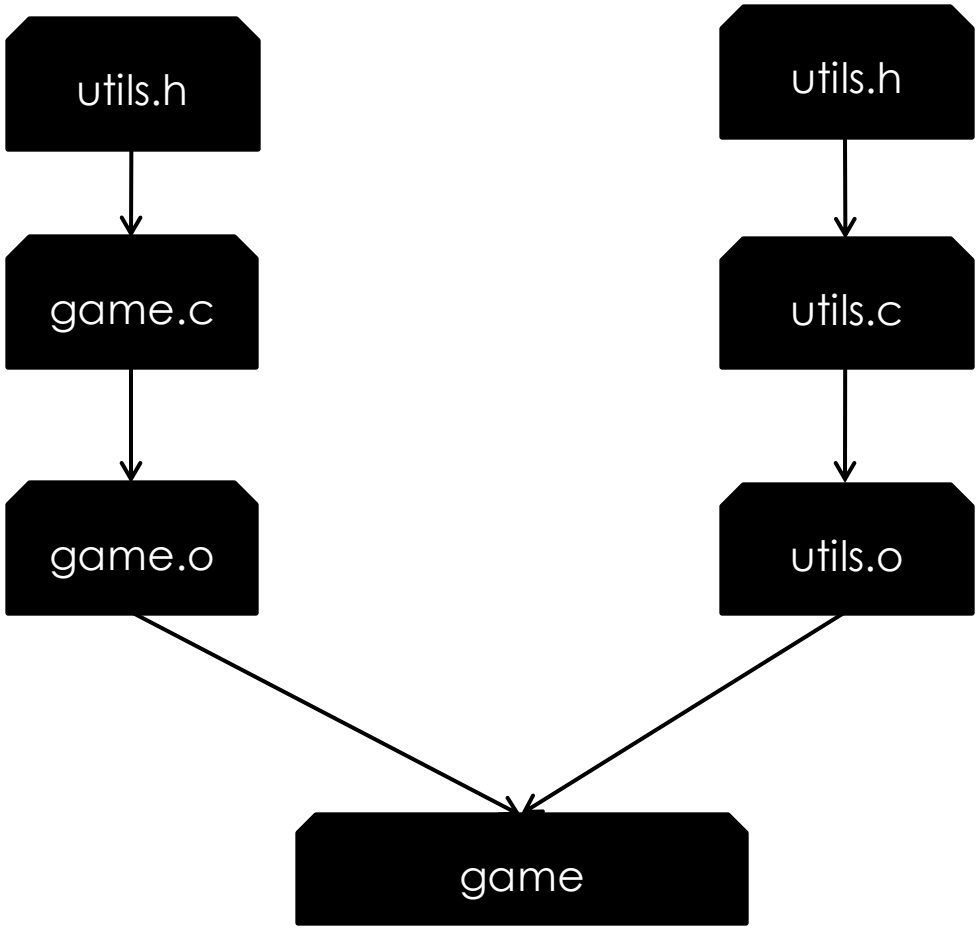
Make par l'exemple

Make dans votre projet

Revenons à nos Moutons



Un petit exemple



Compilation de Plusieurs Fichiers

```
$> gcc game.c –o game.o
$> gcc utils.c -o utils.o
$> gcc game.o utils.o –o game
$> ./game

Et même :
$> gcc game.c utils.c –o game
```

Header (.h) sont dans le même répertoire que les fichiers sources (.c)
Pas de librairies particulières
Pas d'options de compilation
Destination des binaires etc.

Ce sont des options que nous pouvons ajouter :

- -Iinclude_path
- -Llibrary_path -lfoo
- -Wall -O3 –g
- …

MAKE ?

Make :

- Automatisation des 4 phases de compilation
- Largement utilisé dans le développement (Linux/Unix)

Si je peux compiler en 1 ligne, pourquoi utiliser Make ?

- Gestion de la complexité
  - Si un fichier … bof
  - Projets complexes, il faut un système de compilation
- Modulaire, claire, facilité la gestion du projet
- Gain de temps : recompilation des fichiers sources qui ont été modifiés et gestion des dépendances (modification d'un fichier .h)
- Automatisation des phases de compilations mais aussi des étapes de préparations, de tests, de nettoyages par exemple
- Uniformité : définir des règles de compilation commune à tous les fichiers

D'autres outils :

- CMake pour construites des fichiers de build natifs
- Gradle : java, kotlin, multiplateformes
- Apache Ant : plutôt ancien, java
- Maven : plutôt ancien, java
- Les IDE intègrent ce type d'outils automatiquement

FONCTIONNEMENT

Un 1er fichier Makefile

```
# Définition des variables
CC = gcc
CFLAGS = -Wall -g

# Cible par défaut
all: game

# Règle pour créer l'exécutable "game"
game: game.o utils.o
	$(CC) $(CFLAGS) -o game game.o utils.o

# Règle pour compiler game.c en game.o
game.o: game.c utils.h
	$(CC) $(CFLAGS) -c game.c

# Règle pour compiler utils.c en utils.o
utils.o: utils.c utils.h
	$(CC) $(CFLAGS) -c utils.c

# Règle pour nettoyer les fichiers générés
clean:
	rm -f *.o game
```

Va chercher la règle all ou la 1ère règle trouvée

```
$> ls Makefile
Makefile
$> make
...
$> ./game
Hello World !
...
$> make clean
```

FONCTIONNEMENT

cible : dépendance(s)
commande
commande

cible : dépendance(s)
<tabulation>commande
<tabulation>commande

CC : compilateur
CFLAGS : des options
Cible :

- all, game, TARGET pour compiler l'ensemble des fichiers et vérifier les dépendances
- clean pour supprimer des fichiers

Vous pouvez également définir des variables plus génériques

- SRC = liste des sources
- TARGET = cible
- LIBS = librairies
- …

Par exemple (extrait d'un Makefile) :

```
SRC = game.c utils.c
TARGET = game

all: $(TARGET)
$(TARGET): $(SRC)
        $(CC) $(CFLAGS) -o $(TARGET) $(SRC)
```

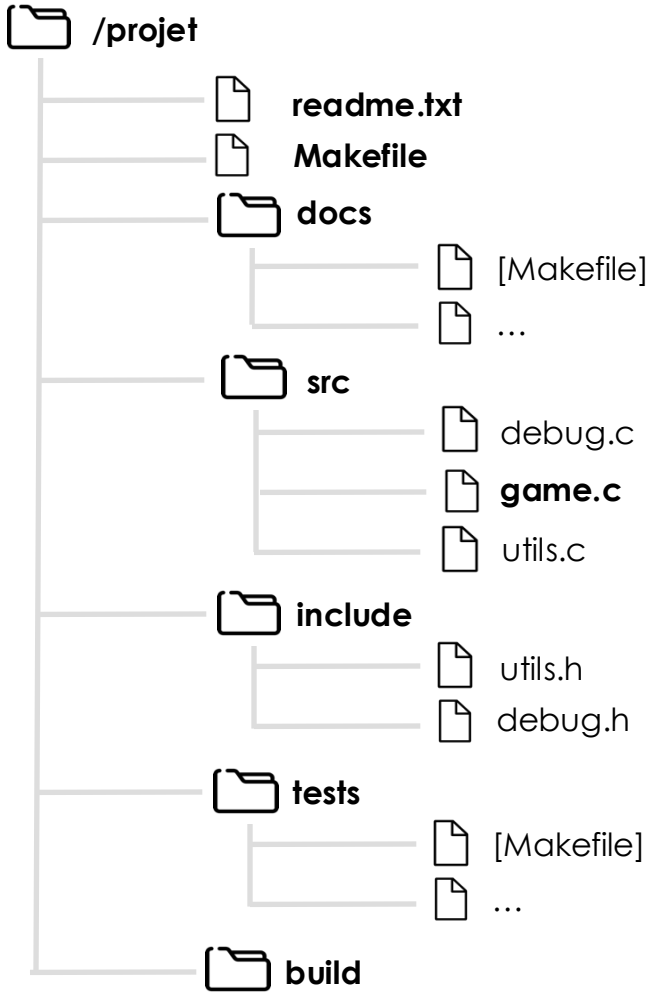
SOMMAIRE

Un rapide point sur la compilation

Make par l'exemple

Make dans votre projet

Structure de Projet Suggérée




```
+-----------------------------------------------
| Compilation
+-----------------------------------------------

Au niveau le plus haut (*)
Compilation, produit l'exécutable ./game
$> make
Tests : compile et exécute les tests sous
./tests
$> make tests
Docs : production des docs sous ./docs
$> make docs

+-----------------------------------------------
| Utilisation
+-----------------------------------------------

$> ./game options (*)
   -l : en mode local (une seule interface)
   -c : IP, port sous la forme hhh.hhh.hhh.hhh:pppp
   -s : port sous la forme numérique pppp
   -ia : ajout d'un ia pour calculer les coups

(*) Dans le respect des consignes y compris le nom
    game de l'exécutable game
```

Travail Demandé Dans le Cadre du Projet

Depuis la racine du dossier de projet la compilation du projet doit être obtenue par :

```
$> make
```

L'exécution du programme doit être obtenue par la ligne de commande conforme aux spécifications et notamment le nom du programme (game)

```
$> ./game [options]
$> ./game -l
        # -l : en mode local (une seule interface)
$> ./game -c hhh.hhh.hhh.hhh:pppp
        # -c : IP:port
$> ./game -s pppp
        # -s : port
$> ./game -ia -c hhh.hhh.hhhh.hhh:pppp
        # IA qui s'occupe de calculer les coups (avec -c)
$> ./game -ia -s pppp
        # IA qui s'occupe de calculer les coups (avec -s)
```

La production et l'exécution des tests sera obtenue par la commande

```
$> make tests
```

La production de la documentation sera obtenue par la commande

```
$> make docs
```

Vers l'Infini et Au-Delà

Variables définies par l'utilisateur

- **nom=valeur (*dynamique*)**
- **nom:=valeur (*statique*)**
- **nom?=valeur (*defaut*)**

Variables internes (dans une commande)

- **$@ le nom de la cible**
- **$< le nom de la première dépendance**
- **$^ la liste énumérée des dépendances (sans duplication)**
- **$+ la liste énumérée des dépendances**
- **$? la liste énumérée des dépendances plus récentes que la cible**
- **$* le nom de la cible sans suffixe**

Des règles implicites pour les sources, les dépendances

VERS L'INFINI ET AU-DELÀ

```
# Variables de Compilation
CC = gcc
CFLAGS = -Wall -g -Iinclude -MMD    # -Iinclude pour indiquer le répertoire des fichiers .h
                                    # -MMD pour générer des fichiers .d de dépendance
TARGET = game

# Répertoires
SRC_DIR = src                       # Répertoire des sources
BUILD_DIR = build                   # Répertoire pour les fichiers objets et dépendances

# Recherche des fichiers sources et génération des listes de fichiers objets et dépendances
SRCS = $(wildcard $(SRC_DIR)/*.c)  # Tous les fichiers .c dans src/
OBJS = $(patsubst $(SRC_DIR)/%.c,$(BUILD_DIR)/%.o,$(SRCS))
                                    # Remplacement src/ par build/ et .c par .o
DEPS = $(OBJS:.o=.d)                # Liste des fichiers de dépendance .d

# Cible par défaut
all: $(TARGET)

# Cible pour créer l'exécutable
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $^

# Inclusion des fichiers de dépendance
-include $(DEPS)

# Règle pour compiler les fichiers .c en fichiers .o
$(BUILD_DIR)/%.o: $(SRC_DIR)/%.c
    $(CC) $(CFLAGS) -c $< -o $@

# Cible pour nettoyer les fichiers générés
.PHONY: clean
clean:
    rm -f $(TARGET) $(BUILD_DIR)/*.o $(BUILD_DIR)/*.d
```

VERS L'INFINI ET AU-DELÀ

```
# Variables de Compilation
CC = gcc
CFLAGS = -Wall -g -Iinclude -MMD

TARGET = game

# Répertoires
SRC_DIR = src
BUILD_DIR = build

# Recherche des fichiers sources et génération des listes de fichiers objets et dépendances
SRCS = $(wildcard $(SRC_DIR)/*.c)
OBJS = $(patsubst $(SRC_DIR)/%.c,$(BUILD_DIR)/%.o,$(SRCS))

DEPS = $(OBJS:.o=.d)

# Cible par défaut
all: $(TARGET)

# Cible pour créer l'exécutable
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $^

# Inclusion des fichiers de dépendanc
-include $(DEPS)

# Règle pour compiler les fichiers .c
$(BUILD_DIR)/%.o: $(SRC_DIR)/%.c
    $(CC) $(CFLAGS) -c $< -o $@

# Cible pour nettoyer les fichiers gé
.PHONY: clean
clean:
    rm -f $(TARGET) $(BUILD_DIR)/*.o
```

-Wall : tous les avertissements de compilation
-g : informations de débogage (-O3 pour optimiser)
-Iinclude : fichier d'entêtes dans le dossier include
- MMD : crée un fichier de dépendances (.d) contenant les dépendances sans inclure les fichiers du système

SRC_DIR : dossier des fichiers sources
BUILD_DIR : le répertoire de *build*
SRCS : recherche automatique des fichiers .c
OBJS : les fichiers objets (patsubst : chemin relatif/absolu)

DEPS : génération des dépendances

$(TARGET) : génération de l'exécutable

-include : inclusion des dépendances

$(BUILD_DIR)/%.o: $(SRC_DIR)/%.c : compilation d'un fichier .c en .o (src → objet) – règle explicite

.PHONY : pour éviter des confusions entre fichier et action
clean : suppression des fichiers générés par la compilation

Un Petit Coup de Main (Non Testé)

```
# Définir le compilateur
CC = gcc

# Utiliser pkg-config pour récupérer les options de compilation et de liaison
CFLAGS = $(shell pkg-config --cflags gtk4)
LIBS = $(shell pkg-config --libs gtk4)

# Nom de l'exécutable
TARGET = game

# Fichier source
SRC = game.c

# Cible par défaut
all: $(TARGET)

# Règle pour créer l'exécutable
$(TARGET): $(SRC)
    $(CC) $(CFLAGS) -o $(TARGET) $(SRC) $(LIBS)

# Cible pour nettoyer les fichiers générés
.PHONY: clean
clean:
    rm -f $(TARGET)
```

A reprendre dans le Makefile précédent

Merci Owen

RÉFÉRENCE

Richard M. Stallman, Roland McGrath et Paul D. Smith. *GNU Make*. 4.4. (https://www.gnu.org/software/make/manual/make.pdf). Free Software Foundation, 2022. isbn : 1-882114-83-3.