

TP 3 : Compilation séparée Makefile, fichiers .h et .c

Objectifs :

- Maîtriser les fichiers de compilation Makefile.
- Maîtriser la compilation séparée.
- Savoir utiliser les bibliothèques prédéfinies en C.

TP 3 : COMPILATION SEPARÉE MAKEFILE, FICHIERS .H ET .C.....	1
COMPILATION SEPARÉE	2
1. INTRODUCTION.....	2
1.1. <i>Comment marche la compilation ?</i>	2
1.2. <i>Intérêt de la compilation séparée</i>	3
1.3. <i>Mise en œuvre</i>	3
2. REGLES.....	3
2.1. <i>Définition</i>	3
2.2. <i>Cible</i>	3
2.3. <i>Dépendances</i>	3
2.4. <i>Commandes</i>	4
3. UN EXEMPLE DE FICHIER MAKEFILE.....	4
3.1. <i>Mon premier Makefile</i>	4
3.2. <i>Comment utiliser mon Makefile</i>	4
4. VARIABLES	4
4.1. <i>Définition et utilisation</i>	4
4.2. <i>Exemple de fichier Makefile avec des variables</i>	5
5. CARACTERES JOCKERS ET VARIABLES AUTOMATIQUES	5
5.1. <i>Caractères jockers</i>	5
5.2. <i>Variables automatiques</i>	5
6. CONVENTIONS DE NOMMAGE	5
6.1. <i>Noms d'exécutables et arguments</i>	5
6.2. <i>Noms des cibles</i>	6
EXERCICE.....	6
ORGANISATION D'UN PROGRAMME.....	6
1. DECOUPAGE D'UN PROGRAMME EN MODULES	7
2. COMPILATION CONDITIONNELLE.....	8
EXERCICES.....	9
EXERCICE 1 : BIBLIOTHEQUE MATHEMATIQUE.....	9
EXERCICE 2 : MOSAÏQUE	9

Compilation séparée

1. Introduction

1.1. Comment marche la compilation ?

La compilation avec gcc s'effectue en quatre étapes pour produire un exécutable :

1. passage au pré-processeur (*preprocessing*) ;
2. compilation en langage assembleur (*compiling*) ;
3. conversion du langage assembleur en code machine (*assembling*) ;
4. édition des liens (*linking*).

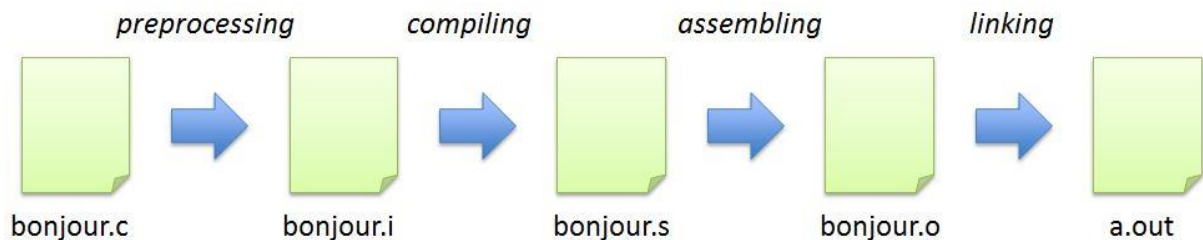


Figure 1 : Chaîne de compilation.

Le prétraitement (*preprocessing*) consiste à transformer le code source d'un programme C en un autre code source C débarrassé des directives de prétraitement et où les macros du processeur ont été expansées. Les directives de prétraitement permettent d'inclure d'autres fichiers contenant des déclarations (ex : `#include <fichier.h>`) et de définir de nouvelles macros (ex : `#define PI 3.14`). Les directives du préprocesseur commencent toutes par un caractère dièse (#).

La compilation (*compiling*) du code C prétraité en assembleur est la phase la plus compliquée. Pendant cette phase, on passe d'un langage de haut niveau vers un langage de très bas niveau : l'assembleur.

L'assemblage (*assembling*) permet la traduction du langage assembleur (texte) en langage machine binaire.

La commande gcc utilisée avec l'option `-c` permet de réaliser ces trois premières étapes. Ainsi, la commande `gcc -c bonjour.c` permet de produire le fichier `bonjour.o` (la commande `gcc -c bonjour.c -o bonjour.o` a le même résultat). Enfin, l'édition de liens (*linking*) permet de combiner éventuellement plusieurs fichiers « objet » produits par la phase d'assemblage ainsi que les éventuelles bibliothèques utilisées (p.ex. pour les entrées sorties) en un seul programme exécutable. Elle fait le lien entre des symboles définis dans certains objets (par exemple des bibliothèques) et ces mêmes symboles utilisés dans d'autres objets. La commande suivante effectue l'édition des liens entre les fichiers objet `bonjour.o` et `bonsoir.o` et produit le fichier exécutable `progexe` : `gcc bonjour.o bonsoir.o -o progexe`. Si le nom de fichier résultant de la compilation n'est pas spécifiée à l'aide de l'option `-o`, il prend par défaut la valeur "a.out".

1.2. Intérêt de la compilation séparée

L'utilité de la compilation séparée est triple :

- la programmation est modulaire, donc plus compréhensible ;
- la séparation en plusieurs fichiers produit des listings plus lisibles ;
- la maintenance est plus facile car seuls les modules modifiés sont recompilés.

La compilation séparée se fait avec la commande `make`¹. Cette commande :

- assure la compilation séparée grâce à la commande `gcc` ;
- utilise des macro-commandes et des variables ;
- permet de ne recompiler que le code modifié ;
- permet d'utiliser des commandes shell.

1.3. Mise en œuvre

La commande `make` requiert pour son fonctionnement un fichier nommé `Makefile`. Le fichier `Makefile` indique à la commande `make` comment exécuter les instructions nécessaires à l'installation d'un logiciel ou d'une bibliothèque. Le fichier `Makefile` doit se trouver dans le répertoire courant lorsque l'on appelle la commande `make` à l'invite du shell. Ce fichier doit être nommé : `Makefile` (avec un M majuscule et le reste en minuscule). Les instructions contenues dans ce fichier doivent respecter une syntaxe particulière expliquée ci-après.

2. Règles

Les fichiers `Makefile` sont structurés grâce aux *règles* ; ce sont elles qui définissent ce qui doit être exécuté et qui permettent de compiler un programme.

2.1. Définition

Une règle est une suite d'instructions qui seront exécutées pour construire *une cible*, mais uniquement si des *dépendances* sont plus récentes (par rapport à la dernière construction de la cible). La syntaxe d'une règle est la suivante :

```
cible : dépendances
      commandes
```

L'espace avant commandes est obligatoire et doit être fait avec une tabulation.

2.2. Cible

La cible est généralement le nom d'un fichier qui va être généré par les commandes qui vont suivre.

2.3. Dépendances

Les dépendances sont les fichiers ou les règles nécessaires à la création de la cible. Par exemple, pour la compilation C, les dépendances peuvent contenir un fichier d'en-tête ou un fichier source.

¹ `make` est essentiel lorsque l'on veut effectuer un portage, car la plupart des logiciels libres UNIX (c'est-à-dire des logiciels qui sont fournis avec le code source) l'utilisent pour leur installation.

2.4. Commandes

Les commandes sont des commandes shell qui seront exécutées au moment de la construction de la cible. La tabulation avant les commandes est obligatoire et si la commande dépasse une ligne, il est nécessaire de signaler la fin de ligne avec un caractère antislash "\".

3. Un exemple de fichier Makefile

3.1. Mon premier Makefile

```
# Mon premier Makefile

all: foobar.o main.o
    gcc -o main foobar.o main.o

foobar.o: foobar.c foobar.h
    gcc -c foobar.c -Wall -ansi -pedantic -o foobar.o

main.o: main.c
    gcc -c main.c -Wall -ansi -pedantic -o main.o
```

Les lignes commençant par le caractère # sont des lignes de commentaires.

Le Makefile ci-dessus a trois règles destinées à construire les cibles `all`, `foobar.o` et `main.o`. La cible principale `all` nécessite les fichiers `foobar.o` et `main.o` pour être exécutée afin d'obtenir le fichier exécutable `main`. Pour obtenir les fichiers `foobar.o` et `main.o`, il faut utiliser les deux cibles suivantes.

3.2. Comment utiliser mon Makefile

La commande `make` permet d'exécuter le fichier Makefile :

```
machine@esisar> make all
```

La commande `make` interprète le fichier Makefile et exécute les commandes contenues dans la règle `all`, une fois que les dépendances `foobar.o` et `main.o` ont été vérifiées. Ce qui signifie, par exemple, que si `foobar.c` ou bien `foobar.h` ont été modifiés depuis la dernière compilation, alors `foobar.o` sera d'abord mis à jour avant que la commande de la règle `all` soit exécutée.

La commande `make` peut être exécutée sans argument. Dans ce cas, elle exécute la première règle rencontrée.

4. Variables

4.1. Définition et utilisation

Les variables doivent être vues comme des macro-commandes (comme `#define` en C). La déclaration se fait ainsi :

```
NOM = VALEUR
```

La valeur affectée à la variable peut comporter n'importe quels caractères. Elle peut être aussi une autre variable.

La syntaxe de l'appel de la variable est la suivante :

`$ (NOM)`

4.2. Exemple de fichier Makefile avec des variables

```
# $(BIN) est le nom du fichier binaire généré
BIN = foo
# $(OBJECTS) sont les objets qui seront générés après la
compilation
OBJECTS = main.o foo.o
# $(CC) est le compilateur utilisé
CC = gcc
# all est la première règle à être exécutée car elle est la
première
# dans le fichier Makefile. Notons que les dépendances peuvent
être
# remplacées par une variable, ainsi que n'importe quelle
chaîne de
# caractères des commandes
all: $(OBJECTS)
    $(CC) $(OBJECTS) -o $(BIN)

main.o: main.c main.h
    $(CC) -c main.c
foo.o: foo.c foo.h main.h
    $(CC) -c foo.c
```

5. Caractères jockers et variables automatiques

5.1. Caractères jockers

Les caractères jockers s'utilisent comme en shell. Les caractères valides sont par exemple : *, ?. L'expression toto?.c représente tous les fichiers commençant par toto et finissant par .c avec une lettre entre ces deux chaînes (toto1.c, toto2.c...). Comme en shell, le caractère \ permet d'inhiber l'action des caractères jockers.

5.2. Variables automatiques

Les variables automatiques sont variables qui sont actualisées au moment de l'exécution de chaque règle en fonction de la cible et des dépendances.

\$@	Nom de la cible
\$<	Première dépendance de la liste des dépendances
\$?	Les dépendances les plus récentes de la cible
\$^	Toutes les dépendances

6. Conventions de nommage

6.1. Noms d'exécutables et arguments

CC	Compilateur C (cc ou gcc)
CXX	Compilateur C++ (c++ ou g++)
RM	Commande pour effacer un fichier (rm)

CFLAGS	Paramètres à passer au compilateur C
CXXFLAGS	Paramètres à passer au compilateur C++

6.2. Noms des cibles

Un utilisateur de make peut donner à ses cibles le nom qu'il désire, mais pour des raisons de lisibilité, on donne toujours un nom standard à certaines cibles :

all	Compile tous les fichiers source pour créer l'exécutable principal
install	Exécute all et copie l'exécutable, les bibliothèques et les fichiers d'en-tête s'il y en a dans le répertoire de destination
uninstall	Détruit les fichiers créés lors de l'installation, mais pas les fichiers du répertoire d'installation où se trouvent les fichiers sources et le Makefile
clean	Détruit tous les fichiers créés par all
dist	Crée un fichier tar de distribution

Exercice

Ecrire un fichier Makefile qui permette de compiler le code de l'exercice du jeu de la multiplication du TP 2.

Organisation d'un programme

Jusqu'à présent, tout votre code était dans un unique fichier source .c. La structure d'un fichier *monProgramme.c* est en général la suivante :

```
// Inclusion de fichiers permettant d'utiliser des fonctions
// de bibliothèques
#include <stdio.h>
#include <stdlib.h>

// Déclarations des constantes
#define TAILLE=100

// Déclarations des variables globales
int entier1 ;
// Déclarations de toutes les fonctions
// Le détail des fonctions se trouve après la fonction main()
void fonction1(void) ;
void fonction2(void) ;

// Programme principal
int main(int argc, char * argv[]) {
...
...
}
```

```
// Détail des fonctions déclarées précédemment
void fonction1(void) {
    ...
}

void fonction2(void) {
    ...
}
```

Cette méthode fonctionne bien dans le cadre de petits programmes assez simples. Dans le cas de plus grosses applications, il est nécessaire de découper le code en fonctions et ces fonctions peuvent être organisées selon leurs préoccupations. Par exemple, pour un logiciel de jeu, il est possible d'avoir des fonctions pour le graphisme et d'autres pour gérer les différents joueurs.

1. Découpage d'un programme en modules

En C, un module (ensemble de fonctions) est réalisé à l'aide de deux sortes de fichiers reconnaissables à leur extension :

- les fichiers .c qui sont des *fichiers source* ;
- les fichiers .h (h pour header en anglais) qui sont les *fichiers d'en-tête*.

Les fichiers .c contiennent l'implémentation des fonctions du module. Les fichiers .h contiennent uniquement les déclarations de type, de fonctions... Le principe généralement appliqué consiste à écrire un fichier .h pour chaque fichier .c et à déclarer dans le fichier .h tout ce qu'exporte le module : lorsqu'un fichier .c ou .h d'un module A utilise un identificateur déclaré dans un module B, alors il inclut le fichier .h de B.

Exemple : découpage en différents fichiers de l'exemple précédent.

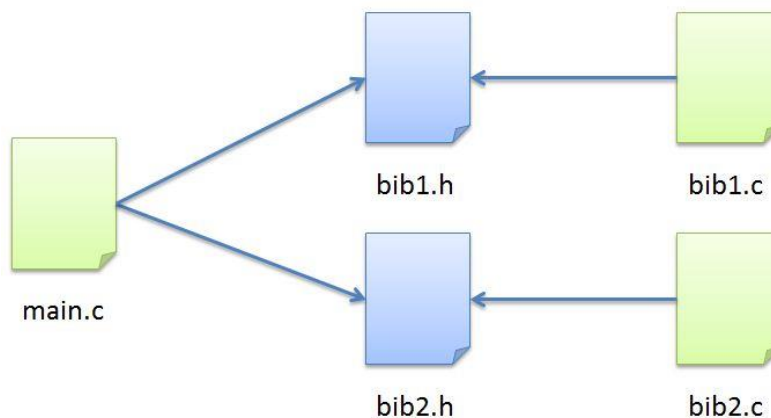


Figure 2 : Architecture du programme.

Les flèches représentent les inclusions.

Les fichiers `bib1.h` et `bib2.h` contiennent les déclarations qui sont utiles au module `bib1` et `bib2` :

Le fichier `bib1.h` :

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

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

```
#define TAILLE=100

void fonction1(void) ;
```

```
int entier1 ;

void fonction2(void) ;
```

Le fichier bib2.h :

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

Les fichiers .c contiennent les implémentations des fonctions :

Le fichier bib1.c

```
#include "bib1.h"

void fonction1(void) {
    ...
}
```

Le fichier bib2.c

```
#include "bib2.h"

void fonction2(void) {
    ...
}
```

Le programme principal ne contient plus que les inclusions nécessaires ainsi que la fonction main().

```
// Inclusion de fichiers permettant d'utiliser des fonctions
// de bibliothèques

#include "bib1.h"
#include "bib2.h"

// Programme principal
int main(int argc, char * argv[]) {
    ...
    ...
}
```

2. Compilation conditionnelle

Avec l'exemple précédent, on remarque que plusieurs bibliothèques identiques ont été incluses dans les différentes déclarations d'inclusion faites dans les différents fichiers (par exemple, la bibliothèque `stdio.h`). Pour éviter de compiler plusieurs fois les mêmes fichiers, on met en place un mécanisme de *compilation conditionnelle*. Le compilateur tiendra compte d'un ensemble de commandes pour savoir ce qui doit être compilé ou non.

Des *directives* permettent d'incorporer ou d'exclure des portions du fichier source dans le texte qui sera analysé par le préprocesseur. Ces directives se classent en deux catégories selon la condition qui régit l'incorporation :

- Existence ou inexistence des symboles ;
- Valeur d'une expression.

Le détail des *directives* sont présentés dans la section 8.2 du document « *Introduction au langage C* » à la page 139.

Pour résumer, les fichiers .h doivent avoir cette forme (illustrée sur l'exemple du fichier `bib1.h`) :

```
#ifndef __BIB1_H__
#define __BIB1_H__
```



```
#include "bibl.h"

void fonction1(void) ;

#endif
```

Exercices

Exercice 1 : Bibliothèque mathématique

Le but de cet exercice est d'écrire une bibliothèque de fonctions mathématiques. Vous devrez structurer et tester correctement votre code (fichiers .h et .c).

- Ecrire une fonction *quotient()* ayant pour paramètres deux entiers positifs a et b et qui retourne le quotient de a et b. Ce calcul doit se faire par soustractions successives (interdiction d'utiliser l'opérateur de division du langage C).
- Ecrire une fonction *reste()*, ayant pour paramètres deux entiers positifs a et b qui retourne le reste de la division de a par b. Cette fonction doit utiliser la fonction *quotient()* (et non pas l'opérateur %).
- Ecrire une fonction *valeurAbsolue()* qui calcule la valeur absolue d'un entier. Cette fonction peut utiliser la fonction *abs()* de la bibliothèque *stdlib.h*.
- Ecrire une fonction *ppcm()* qui calcule le plus petit commun multiple. Pour rappel, il est possible de le calculer ainsi :

$$ppcm(a, b) = \frac{|ab|}{pgcd(a, b)}$$

Exercice 2 : Mosaïque

Le but de cet exercice est d'utiliser une bibliothèque graphique afin de tracer un carré puis une mosaïque constituée de carrés.

La bibliothèque contient des fonctions qui permettent :

- d'ouvrir et d'initialiser une fenêtre pour dessiner : *gr_open_w()* ;
- de fermer une fenêtre ouverte : *gr_close()* ;
- de changer la couleur du pinceau : *set_blue()*, *set_red()*, *set_green()*, *set_yellow()*, *set_black()* ;
- de tracer une ligne : *line()* ;
- de récupérer les coordonnées d'un clic souris (cette commande attend qu'un clic se produise) : *clicquer_xy()* ;
- ...

A partir de ces fonctions, vous devrez tracer un carré « sur la pointe » :

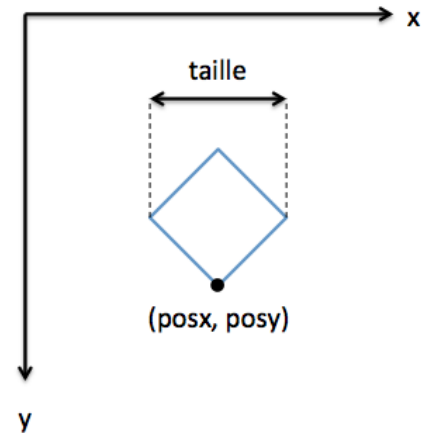
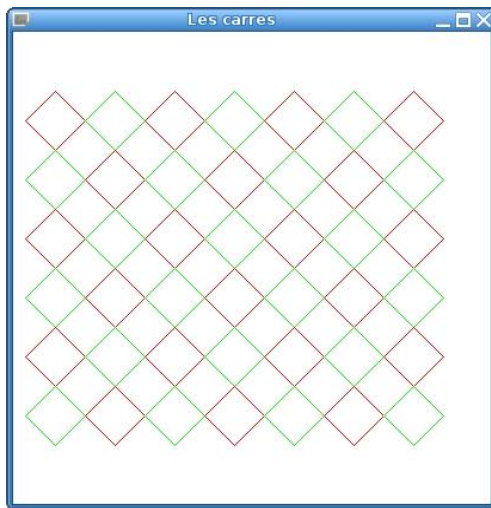


Figure 3 : Mosaïque.

1. Ecrire le fichier *Makefile* qui permet de compiler les fichiers sources qui vous sont fournis. L'utilisation de la bibliothèque graphique nécessite d'utiliser en plus des options de compilation habituelles (`-Wall`, `-ansi`, `-pedantic`) les options `-lX11` et `-L/usr/X11R6/lib`.

Pour la suite, vous devez créer deux fichiers (*dessine.h* et *dessine.c*) dans lesquels vous allez déclarer et implémenter les fonctions suivantes qui seront testées dans le fichier *main.c* :

2. Ecrire une fonction `dessineCarre()` qui dessine un carré « sur la pointe ». Cette fonction prend en paramètre les coordonnées d'un point ainsi que la taille du carré.
3. Ecrire une fonction `dessineCarreDiagonale()` qui dessine un carré ainsi que ses diagonales. Cette fonction prend en paramètre les coordonnées d'un point ainsi que la taille du carré.
4. Ecrire une fonction `dessineMosaique()` qui dessine une mosaïque comme l'illustre la Figure 3. Cette fonction prend en paramètre la taille des carrés, la position de la mosaïque, ses largeur et hauteur en nombre de carrés :

