# Guide d'utilisation de `make`

D. Renault

29 novembre 2022

Version 2.0

**Résumé**

Le but de ce document est de donner une introduction à `make`, un outil d'aide à la compilation, ou *build automation tool*. Il détaille les principes généraux de la gestion du processus de compilation, et fournit des exemples à travers le programme `make`.

## Qu'est-ce qu'un outil d'aide à la compilation ?

La présence de code *source* dans un projet implique naturellement l'existence d'un processus permettant de transformer cette source en un produit fini, processus usuellement nommé *compilation*. Le terme de compilation est ici utilisé au sens large : il peut s'agir de la génération d'un exécutable à partir de fichiers source C, d'un fichier `pdf` à partir d'un code LaTeX ou d'un fichier image au format `png` à partir de sa description au format `pov` par un logiciel de lancer de rayons comme `povray` [1]. En fait, les commandes de compilation sont la plupart du temps spécifiques au projet, et peuvent rapidement devenir nombreuses et complexes. Aussi, l'ordre dans lequel elles doivent s'exécuter n'est pas forcément simple à calculer. Pour gérer ce type de problèmes, il s'avère utile de disposer d'un outil d'assistance à la compilation, ou *build automation tool*, qui permet d'organiser et d'automatiser le processus de compilation. La famille la plus utilisée de tels outils est la famille `make`, contenant en particulier BSD `make` et GNU `make` [2].

Noter que le processus de compilation n'est qu'un maillon au milieu de la chaîne de production du logiciel : à partir d'un code source générique, il s'avère généralement nécessaire d'adapter les sources à l'environnement logiciel [3] (*configuration*), de génèrer les binaires associés sous forme d'exécutables et de bibliothèques (*compilation*), et finalement d'installer ces binaires au sein de l'environnement logiciel (*installation*). Pour prendre en charge ces processus, il existe des outils nommés *build systems* dont le rôle est d'automatiser la chaîne précédente, et ainsi de favoriser la portabilité du coude source. La phase de compilation d'un build system peut d'ailleurs s'appuyer sur un outil comme `make` – c'est le cas des outils GNU comme `automake`, ainsi que de `cmake` – ou bien s'y substituer entièrement

1. `http://www.povray.org`
2. `https://www.gnu.org/software/make`
3. L'exemple le plus courant est celui de l'*endianness*, qui influe sur le stockage des entiers en mémoire, en commençant soit par le bit de poids fort (*big-endian*), soit par le bit de poids faible (*little-endian*).

– comme pour maven (Java) et scons (Python). Nous ne nous intéresserons ici qu'à la partie « compilation » de la chaîne de production du logiciel.

Dans ce document sont expliqués les concepts généraux et les modalités de fonctionnement de GNU make, un outil d'assistance à la compilation, notamment utilisé à l'ENSEIRB pour la gestion des projets en 1ère année.

# 1 Principes de base

À l'ENSEIRB-Matmeca, le chemin d'accès à la commande GNU make est :

```
make
```

La commande make est normalement installée par défaut dans les distributions récentes.

## 1.1 Fichier `Makefile`

Le programme make utilise un fichier de configuration nommé Makefile [4], qui contient les informations permettant de compiler un projet. Lorsque la commande make est utilisée dans un répertoire donné, elle recherche un fichier Makefile dans ce répertoire, et lance une série de commandes de compilations décrites dans ce fichier. Il est possible de placer un fichier Makefile dans chaque répertoire d'un projet donné. Voici un exemple simple :

```
all:
	gcc -std=c99 -lm fichier.c -o main
```


Il est impératif de ne pas oublier les tabulations (le caractère noté « ⇥ ») dans les fichiers Makefile, sous peine de rejet par la commande make.

Avec un tel fichier Makefile dans le répertoire courant, la commande make se charge d'exécuter la commande de compilation apparaissant dans la seconde ligne :

```
sh% make
gcc -std=c99 -lm fichier.c -o main
sh%
```

L'utilisation de make permet ici de définir la phase de compilation en la spécifiant sous la forme d'une liste de commandes à exécuter, stockées dans le fichier Makefile.

→ L'utilisation de make a un intérêt en terme de *portabilité* : en transmettant ce fichier avec le code source, le programmeur transmet une description du processus de compilation de son projet.

→ Toute personne recevant un code source contenant un fichier Makefile dispose quant à elle d'un *moyen générique* pour compiler ce code avec la commande make (qu'il s'agisse d'un projet C, LaTeX ou autre ...)

4. La majuscule est importante !

## 1.2 Règle

Un fichier `Makefile` est principalement constitué d'une suite de règles. Une règle permet de décrire un ensemble de commandes à appliquer pour effectuer une tâche simple du processus de compilation (comme par exemple compiler un unique fichier). Chaque règle d'un `Makefile` se présente sous la forme suivante :

```
cible: prerequis_1 ... prerequis_m
	commande_1
	...
	commande_n
```

Une règle se compose des trois parties suivantes :

- une **cible** : il s'agit soit d'un mot-clé (par exemple "`all`"' pour la règle principale, ou "`clean`" pour nettoyer le répertoire courant), soit d'un nom de fichier qui est produit par la règle ("`fichier.o`" lorsqu'il s'agit de compiler "`fichier.c`").
- une liste de **commandes** qui doivent être exécutées lorsque le programme `make` doit atteindre la cible courante.
- une liste de **prérequis** : il s'agit d'une liste de cibles qui doivent être atteintes avant de pouvoir exécuter la cible courante.

Une règle est dite « à jour » lorsque :

1. sa cible correspond effectivement à un fichier existant dans le répertoire courant,
2. et si sa cible est plus récente que tous ses prérequis (selon la date de dernière modification des fichiers).

Par exemple, la règle suivante permet de compiler un fichier LaTeX :

```
rapport.ps: rapport.tex
	latex rapport.tex && dvips rapport.dvi -o
```

Cette règle a pour but de générer le fichier `rapport.ps`. Elle possède comme prérequis le fichier `rapport.tex`, qui ne correspond à aucune règle dans le `Makefile`. Si la date de dernière modification du fichier `rapport.tex` est plus récente que la date de dernière modification de `rapport.ps`, alors la règle n'est pas à jour. Lorsqu'on lui demande de la mettre à jour, `make` applique la liste de commandes associée pour compiler le fichier LaTeX.

## 1.3 Fonctionnement de `make`

Exécutons la commande suivante dans la ligne de commande :

`make` ⟨`target`⟩

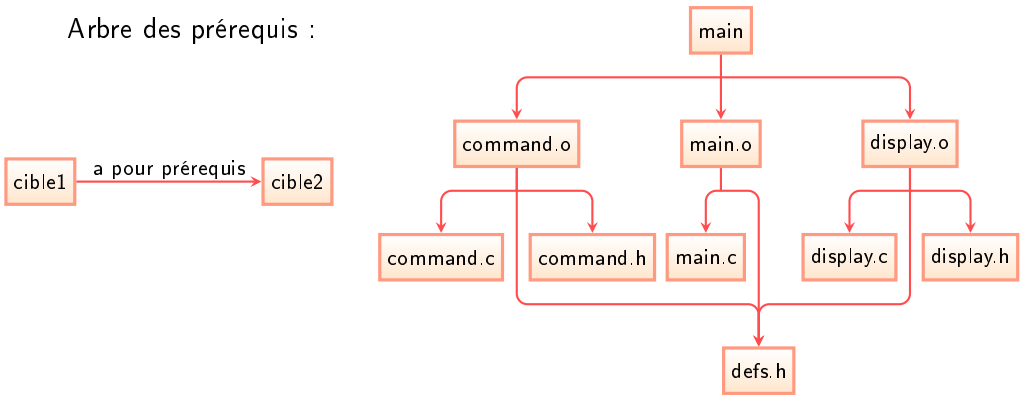
Ici, ⟨`target`⟩ est soit vide, soit la cible d'une règle définie dans le fichier `Makefile` courant. Le programme `make` applique alors le mécanisme suivant :

1. Premièrement, le programme `make` recherche la règle spécifiée par la cible ⟨`target`⟩ dans le fichier `Makefile` se trouvant dans le répertoire courant. Si l'on ne spécifie pas de cible à atteindre, alors `make` considère la première règle du fichier.
2. Ensuite, `make` construit la liste des prérequis associés à cette règle : si la règle ne comporte aucun prérequis, `make` exécute la liste de commandes associées à cette règle ; sinon, `make` s'applique récursivement sur chacun des prérequis.
3. Enfin, lorsque la règle n'est pas à jour, `make` applique la liste des commandes associée.

Considérons à titre d'exemple le `Makefile` suivant, et construisons les relations de dépendances (données par les prérequis) sous la forme d'un arbre :

```
1  main: main.o command.o display.o
2  ␣␣␣␣␣␣␣␣ gcc display.o command.o main.o -o main
3
4  main.o: main.c defs.h
5  ␣␣␣␣␣␣␣␣ gcc -c main.c
6  command.o: command.c defs.h command.h
7  ␣␣␣␣␣␣␣␣ gcc -c command.c
8  display.o: display.c defs.h display.h
9  ␣␣␣␣␣␣␣␣ gcc -c display.c
10
11 clean:
12 ␣␣␣␣␣␣␣␣ rm main main.o command.o display.o
```

Arbre des prérequis :




Supposons d'abord qu'aucune des règles n'est à jour. Afin d'atteindre la cible `main`, `make` construit la liste des cibles qui doivent préalablement être atteintes, dans notre cas `command.o`, `display.o` et `main.o`. Ensuite, `make` s'exécute récursivement pour appliquer les trois règles correspondantes. La règle `clean` n'est pas appliquée, puisqu'elle n'apparaît pas comme prérequis d'une autre règle.

Supposons maintenant que toutes les règles soient à jour. Si l'on modifie le fichier `defs.h`, alors les trois règles `.o` ne seront plus à jour. L'application de la commande `make` va alors consister à atteindre les cibles `.o` correspondant aux prérequis, vérifier qu'elles ne sont plus à jour, les recompiler, puis terminer en appliquant la commande de compilation associée à la règle par défaut (ici `main`).

Enfin, la règle `clean` ne correspond (en temps normal) à aucun fichier dans le répertoire courant. Selon la définition, cette règle ne peut jamais être à jour. Cela signifie que la commande `make clean` appliquera systématiquement la règle, quels que soient les fichiers dans le répertoire courant [5].

# 2 Utilisation de variables

L'exemple donné dans le paragraphe précédent montre à quel point un `Makefile` avec peu de fichier peut très rapidement devenir verbeux au point de le rendre confus. Afin d'augmenter la lisibilité des `Makefile`, mais aussi la réutilisabilité des règles, il est possible d'utiliser des variables au même titre que dans un script shell.

## 2.1 Variables globales

La définition d'une variable globale dans un `Makefile` s'écrit :

$$\langle \texttt{name} \rangle = \langle \texttt{value} \rangle$$

Ensuite, pour utiliser cette variable, il suffit de l'écrire ainsi : `$(⟨name⟩)`. Classiquement, les variables permettent de représenter de manière concise et uniforme les chemins d'accès des exécutables (comme le compilateur `CC`), les drapeaux de compilation (`CFLAGS`) ou encore toute information devant être recopiée plusieurs fois à l'intérieur d'un `Makefile`.

```
1 CC = gcc
2 CFLAGS = -std=c99
3 OBJS = main.o command.o display.o
4
5 all: $(OBJS)
6 	$(CC) $(CFLAGS) $(OBJS) -o main
```


Les variables ne sont pas étendues (*expanded*) lors de leur définition. En particulier, le caractère "`*`" ne représente pas l'ensemble des fichiers du répertoire.

## 2.2 Règles génériques / Pattern rules

Certaines règles de compilation ont tendance à se ressembler fortement. C'est le cas notamment des règles permettant de produire un fichier `.o` à partir d'un fichier `.c`, comme on peut le voir dans le `Makefile` en page précédente. Il existe un moyen de créer des règles *génériques*, dont le comportement ne dépend plus directement du fichier, mais est spécifié à l'aide de variables.

Concrètement, une règle générique est composée de :

5. Cf. aussi plus loin l'utilistion de `.PHONY`.

- Une cible contenant le caractère "%", qui sert de *wildcard* au même titre que le caractère "*" dans les scripts shell. Le mot représenté par ce caractère est appelé la racine (*stem*).
- Des prérequis pouvant aussi contenir le caractère "%", qui est alors censé représenter la même racine que dans la cible.
- Une liste de commandes qui réutilisent la racine à l'aide de variables spéciales.

Par exemple, la règle suivante permet de compiler tous les fichiers dans le répertoire courant dont le nom se termine par ".c".

```
1 CC = gcc
2 CFLAGS = -std=c99
3 OBJS = main.o command.o display.o
4
5 main : $(OBJS)
6 ␣␣␣␣␣␣␣ $(CC) $(CFLAGS) $(OBJS) -o main
7
8 %.o : %.c
9 ␣␣␣␣␣␣␣ $(CC) -c $(CFLAGS) $< -o $@
```

Quelques exemples de variables spéciales utilisables dans les commandes :

- $* : la chaîne de caractères correspondant à la racine
- $@ : le nom entier de la cible
- $< : le nom du premier prérequis
- $^ : la liste des prérequis sans doublons

## 2.3 Substitution dans les variables

Certains Makefile demandent de pouvoir faire des substitutions dans les noms des variables utilisées, à la manière de ce qui peut se faire avec la commande sed. A cet effet, la transformation suivante permet de le faire directement pour une variable d'un Makefile :

$(⟨variable⟩:⟨pattern1⟩=⟨pattern2⟩)

```
1 SOURCES = main.c command.c display.c
2
3 clean:
4 ␣␣␣␣␣␣␣ rm -f $(SOURCES:.c=.o)
```

Néanmoins, il est parfois nécessaire de faire des substitutions directement dans les variables, ce qui peut se faire avec les fonctions de manipulation de texte de make [6] comme dans l'exemple suivant où la fonction patsubst est utilisée pour transformer une liste de fichiers source .c en liste d'objets .o :

```
1 SOURCES = main.c command.c display.c
2 OBJS = $(patsubst %.c,%.o,$(SOURCES))
3
4 all: $(OBJS)
5         gcc $(CFLAGS) $(OBJS) src/project.c -o project
```

6. https://www.gnu.org/software/make/manual/html_node/Text-Functions.html

# 3 Principes avancés

## 3.1 Exemples de `Makefile` classiques

- Règle `clean` :
La règle de nettoyage des fichiers dans le répertoire courant se nomme usuellement `clean`. L'utilisation de `.PHONY` assure le fonctionnement de cette règle, même s'il existe un fichier nommé `clean` dans le répertoire courant.

```
SOURCES = main.c command.c
CLEANOBJ = $(SOURCES:.c=.o)

clean:
        rm -f $(CLEANOBJ)

.PHONY: clean
```

- Simple `Makefile` pour projet `C` :
La règle de transformation des fichiers `.c` en `.o` fait partie des règles prédéfinies dans `make`[7], ce qui fait qu'il n'est pas forcément nécessaire de la redéfinir.

```
CC=gcc
CFLAGS=-c -Wall -std=c99
SOURCES=main.c command.c display.c
OBJS=$(SOURCES:.c=.o)
EXECUTABLE=main

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJS)
        $(CC) $(OBJS) -o $@

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

- Simple `Makefile` pour projet LaTeX :
Ce `Makefile` montre les limites du programme `make` : il arrive souvent que `latex` demande de compiler plusieurs fois son projet. Ici, `make` n'autorisera qu'une seule recompilation du fichier `.tex` avant de considérer la règle comme à jour. Une solution sous-optimale consiste à compiler deux fois le fichier à l'aide de `latex`.

```
PROJECT = rapport

all: $(PROJECT).pdf

$(PROJECT).pdf: $(PROJECT).tex
        pdflatex $<

clean:
        rm -f *~ *.aux *.log *.dvi
```

## 3.2 Gestion des dépendances en C

Un problème général lors de l'utilisation de fichiers de code est la gestion des dépendances entre les fichiers. Usuellement, un fichier `.o` dépend au moins du fichier `.c` correspondant, mais il peut s'avérer intéressant d'inclure dans les dépendances l'ensemble des fichiers nécessaires à la compilation du `.o` (au moins pour que `make` sache à quel moment il est nécessaire de recompiler un fichier ou pas).

Le compilateur `gcc` propose deux options de compilation nommées `-M` et `-MM` qui renvoient directement une règle pour `make` associée à un fichier source donné :

```
sh% gcc -MM automaton.c
automaton.o: automaton.c automaton.h state.h letter.h
sh%
```

7. Cf. https://www.gnu.org/software/make/manual/html_node/Implicit-Rules.html

Il devient alors aisé de construire une règle qui applique cette commande à l'ensemble des fichiers source dans le répertoire courant, et génère un fichier nommé `.depend` qui contienne toutes ces règles. La directive `-include` est ensuite utilisée pour que le fichier `Makefile` inclue le fichier `.depend` généré.

```
1 SOURCES = main.c command.c display.c
2
3 .depend : $(SOURCES)
4         $(CC) $(CFLAGS) -MM $(SOURCES)
5
6 %.o:
7         $(CC) -c $(CFLAGS) $< -o $@
8
9 -include .depend
```

Faire attention à enlever les dépendances à la règle générique "`%.o`", sinon elle prendrait précédence sur les dépendances générées par `gcc`.

Cette procédure demande à ce que le fichier de dépendances soit généré avant les commandes de compilation, et à chaque fois que de nouvelles dépendances sont rajoutées dans les fichiers source. Elle utilise le fait que les prérequis pour une règle donnée peuvent être écrits en plusieurs fois dans le `Makefile`. Cela permet d'utiliser une règle générique pour compiler les fichiers `.o`, tout en spécifiant des prérequis spécifiques à chaque fichier `.c`.

### 3.3 Débugguer les appels à `make`

Dans certaines situations, il n'est pas évident comment `make` applique les règles fournies dans le `Makefile`. Il est néanmoins possible de demander à `make` d'afficher l'ensemble des règles dans sa base de données :

```
make -p
make --print-data-base
```

La base en question est généralement très fournie, parce qu'elle contient l'ensemble des règles implicites de `make`, mais assez facile à fouiller (par exemple avec un pager comme `less`). Dans l'exemple suivant, pour générer `neighbors.o`, on peut remarquer la liste des dépendances (l. 1), le fait qu'une règle implicite est utilisée (l. 2-3) et la recette à appliquer (l. 7-8) :

```
1 neighbors.o: src/neighbors.c src/neighbors.h src/geometry.h
2 #  Implicit rule search has been done.
3 #  Implicit/static pattern stem: 'neighbors'
4 #  Last modified 2022-11-28 18:14:51.536872722
5 #  File has been updated.
6 #  Successfully updated.
7 #  recipe to execute (from 'Makefile', line 24):
8         $(CC) -c $(CFLAGS) $< -o $@
```

## Bibliographie


— La page de développement de GNU `make` : `http://www.gnu.org/software/make/`
— La documentation de GNU `make` : `http://www.gnu.org/software/make/manual/`
— Un tutorial sur `make` : `https://makefiletutorial.com`