

# Introduction à Makefile

par gl

Date de publication : 04/04/2004

Dernière mise à jour : 08/10/2005



- Avant-propos
- Remerciement
- Avertissement
- 1 - Le projet exemple : "hello world"
- 2 - Présentation de Makefile
  - 2.1 - Makefile minimal
  - 2.2 - Makefile enrichi
- 3 - Définition de variables
  - 3.1 - Variables personnalisées
  - 3.2 - Variables internes
- 4 - Les règles d'inférence
- 5 - La cible .PHONY
- 6 - Génération de la liste des fichiers objets
- 7 - Construction de la liste des fichiers sources
- 8 - Commandes silencieuses
- 9 - Les Makefiles conditionnels
- 10 - sous-Makefiles
- 11 - Bibliographie et liens

## Avant-propos

Les Makefiles sont des fichiers, généralement appelés makefile ou Makefile, utilisés par le programme make pour exécuter un ensemble d'actions, comme la compilation d'un projet, l'archivage de document, la mise à jour de site, etc. Cet article présentera le fonctionnement de makefile au travers de la compilation d'un petit projet en C.

## Remerciement

Merci à **Alacazam**, **Anomaly**, et à **Pomalaix** pour la correction orthographique.

Merci également à **Nyal**, **Emmanuel Delahaye**, **\_Anne\_** et **le mage tophinus** pour leurs suggestions et corrections.

## Avertissement

Il existe une multitude d'utilitaires de Makefile fonctionnant sur différents systèmes (gmake, nmake, tmake, etc.). Les Makefiles n'étant malheureusement pas normalisés, certaines syntaxes ou fonctionnalités peuvent ne pas fonctionner sur certains utilitaires.

Le présent article se base sur l'utilitaire GNU make. Toutefois les notions abordées devraient être utilisables avec la majorité des utilitaires.

## 1 - Le projet exemple : "hello world"

Le projet utilisé ici sera un classique "hello world" découpé en trois fichiers:

hello.c

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

void Hello(void)
{
    printf("Hello World\n");
}
```

hello.h

```
#ifndef H_GL_HELLO
#define H_GL_HELLO

void Hello(void);

#endif
```

main.c

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

int main(void)
{
    Hello();
    return EXIT_SUCCESS;
}
```

## 2 - Présentation de Makefile

Un Makefile est un fichier constitué de plusieurs règles de la forme :

```
cible: dependance
commandes
```

Chaque commande est précédée d'une tabulation.

Lors de l'utilisation d'un tel fichier via la commande make la première règle rencontrée, ou la règle dont le nom est spécifié, est évaluée. L'évaluation d'une règle se fait en plusieurs étapes :

- Les dépendances sont analysées, si une dépendance est la cible d'une autre règle du Makefile, cette règle est à son tour évaluée.
- Lorsque l'ensemble des dépendances est analysé et si la cible ne correspond pas à un fichier existant ou si un fichier dépendance est plus récent que la règle, les différentes commandes sont exécutées.

### 2.1 - Makefile minimal

Le Makefile minimal du projet est donc :

#### Makefile

```
hello: hello.o main.o
    gcc -o hello hello.o main.o

hello.o: hello.c
    gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic

main.o: main.c hello.h
    gcc -o main.o -c main.c -W -Wall -ansi -pedantic
```

Regardons de plus près sur cet exemple comment fonctionne un Makefile :

Nous cherchons à créer le fichier exécutable hello, la première dépendance est la cible d'une des règles de notre Makefile, nous évaluons donc cette règle. Comme aucune dépendance de hello.o n'est une règle, aucune autre règle n'est à évaluer pour compléter celle-ci.

Deux cas se présentent ici : soit le fichier hello.c est plus récent que le fichier hello.o, la commande est alors exécutée et hello.o est construit, soit hello.o est plus récent que hello.c est la commande n'est pas exécutée. L'évaluation de la règle hello.o est terminée.

Les autres dépendances de hello sont examinées de la même manière puis, si nécessaire, la commande de la règle hello est exécutée et hello est construit.

### 2.2 - Makefile enrichi

Plusieurs cas ne sont pas gérés dans l'exemple précédent :

- Un tel Makefile ne permet pas de générer plusieurs exécutables distincts.
- Les fichiers intermédiaires restent sur le disque dur même lors de la mise en production.
- Il n'est pas possible de forcer la régénération intégrale du projet

Ces différents cas conduisent à l'écriture de règles complémentaires :

- `all` : généralement la première du fichier, elle regroupe dans ces dépendances l'ensemble des exécutables à produire.
- `clean` : elle permet de supprimer tout les fichiers intermédiaires.
- `mrproper` : elle supprime tout ce qui peut être régénéré et permet une reconstruction complète du projet.

En ajoutant ces règles complémentaires, notre Makefile devient donc :

#### Makefile

```
all: hello

hello: hello.o main.o
    gcc -o hello hello.o main.o

hello.o: hello.c
    gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic

main.o: main.c hello.h
    gcc -o main.o -c main.c -W -Wall -ansi -pedantic

clean:
    rm -rf *.o

mrproper: clean
    rm -rf hello
```

## 3 - Définition de variables

### 3.1 - Variables personnalisées

Il est possible de définir des variables dans un Makefile, ce qui rend les évolutions bien plus simples et plus rapides, en effet plus besoin de changer l'ensemble des règles si le compilateur change, seule la variable correspondante est à modifier.

Une variable se déclare sous la forme `NOM=VALEUR` et se voir utiliser via `$(NOM)`.

Nous allons donc définir quatre variables dans notre Makefile :

- Une désignant le compilateur utilisée nommée `CC` (une telle variable est typiquement nommé `CC` pour un compilateur C, `CPP` pour un compilateur C++).
- `CFLAGS` regroupant les options de compilation (Généralement cette variable est nommée `CFLAGS` pour une compilation en C, `CXXFLAGS` pour le C++).
- `LDFLAGS` regroupant les options de l'édition de liens.
- `EXEC` contenant le nom des exécutables à générer.

Nous obtenons ainsi :

#### Makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
    $(CC) -o hello hello.o main.o $(LDFLAGS)

hello.o: hello.c
    $(CC) -o hello.o -c hello.c $(CFLAGS)

main.o: main.c hello.h
    $(CC) -o main.o -c main.c $(CFLAGS)

clean:
    rm -rf *.o

mrproper: clean
    rm -rf $(EXEC)
```

### 3.2 - Variables internes

Il existe plusieurs variables internes au Makefile, citons entre autres :

<code>\$@</code>	Le nom de la cible
<code>\$&lt;</code>	Le nom de la première dépendance
<code>\$\$^</code>	La liste des dépendances

\$?	La liste des dépendances plus récentes que la cible
\$*	Le nom du fichier sans suffixe

Notre Makefile ressemble donc maintenant à :

#### Makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
$(CC) -o $@ $^ $(LDFLAGS)

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

main.o: main.c hello.h
$(CC) -o $@ $< $(CFLAGS)

clean:
rm -rf *.o

mrproper: clean
rm -rf $(EXEC)
```

## 4 - Les règles d'inférence

Makefile permet également de créer des règles génériques (par exemple construire un `.o` à partir d'un `.c`) qui se verront appelées par défaut.

Une telle règle se présente sous la forme suivante :

```
%.o: %.c
commandes
```

Il existe une autre notation plus ancienne de cette règle :

```
.c.o:
commandes
```

Il devient alors aisé de définir des règles par défaut pour générer nos différents fichiers

### Makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
$(CC) -o $@ $^ $(LDFLAGS)

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

clean:
rm -rf *.o

mrproper: clean
rm -rf $(EXEC)
```

Comme le montre clairement l'exemple précédent, `main.o` n'est plus reconstruit si `hello.h` est modifié. Il est possible de préciser les dépendances séparément des règles d'inférence et de rétablir le fonctionnement original, pour obtenir finalement :

### Makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
$(CC) -o $@ $^ $(LDFLAGS)
```

### Makefile

```
main.o: hello.h

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

clean:
rm -rf *.o

mrproper: clean
rm -rf $(EXEC)
```

## 5 - La cible .PHONY

Dans l'exemple présent, clean est la cible d'une règle ne présentant aucune dépendance. Supposons que clean soit également le nom d'un fichier présent dans le répertoire courant, il serait alors forcément plus récent que ses dépendances et la règle ne serait alors jamais exécutée.

Pour pallier ce problème, il existe une cible particulière nommée .PHONY dont les dépendances seront systématiquement reconstruites.

Nous obtenons donc :

### Makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
    $(CC) -o $@ $^ $(LDFLAGS)

main.o: hello.h

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

.PHONY: clean mrproper

clean:
    rm -rf *.o

mrproper: clean
    rm -rf $(EXEC)
```

## 6 - Génération de la liste des fichiers objets

Plutôt que d'énumérer la liste des fichiers objets dans les dépendances de la règle de construction de notre exécutable, il est possible de la générer automatiquement à partir de la liste des fichiers sources. Pour cela nous rajoutons deux variables au Makefile :

- SRC qui contient la liste des fichiers sources du projet.
- OBJ pour la liste des fichiers objets.

La variable OBJ est remplie à partir de SRC de la manière suivante :

```
OBJ= $(SRC:.c=.o)
```

Pour chaque fichier .c contenu dans SRC nous ajoutons à OBJ un fichier de même nom mais portant l'extension .o.

Nous obtenons alors :

### Makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
SRC= hello.c main.c
OBJ= $(SRC:.c=.o)

all: $(EXEC)

hello: $(OBJ)
$(CC) -o $@ $^ $(LDFLAGS)

main.o: hello.h

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

.PHONY: clean mrproper

clean:
rm -rf *.o

mrproper: clean
rm -rf $(EXEC)
```

## 7 - Construction de la liste des fichiers sources

De la même manière, il peut être utile de gérer la liste des fichiers sources de manière automatique (attention toutefois à la gestion des dépendances vis à vis des fichiers d'entête). Pour ce faire nous allons faire remplir la variable SRC avec les différents fichiers .c du répertoire.

La première idée qui vient généralement pour réaliser cette tâche est l'utilisation du joker \*.c, malheureusement il n'est pas possible d'utiliser ce joker directement dans la définition d'une variable. Nous devons utiliser la commande wildcard qui permet l'utilisation des caractères joker.

Le Makefile correspondant est :

### Makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)

all: $(EXEC)

hello: $(OBJ)
$(CC) -o $@ $^ $(LDFLAGS)

main.o: hello.h

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

.PHONY: clean mrproper

clean:
rm -rf *.o

mrproper: clean
rm -rf $(EXEC)
```

## 8 - Commandes silencieuses

Lorsque le nombre de règles d'un Makefile augmente, il devient très rapidement fastidieux de trouver les messages d'erreur affichés au milieu des lignes de commandes. Les Makefiles permettent de désactiver l'écho des lignes de commandes en rajoutant le caractère @ devant la ligne de commande, notre Makefile devient alors :

### Makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

all: $(EXEC)

hello: $(OBJ)
    @$ (CC) -o $@ $^ $(LDFLAGS)

main.o: hello.h

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

.PHONY: clean mrproper

clean:
    @rm -rf *.o

mrproper: clean
    @rm -rf $(EXEC)
```

## 9 - Les Makefiles conditionnels

Les Makefiles nous offrent en plus une certaine souplesse en introduisant des directives, assez proches des directives de compilation du C, qui permettent d'exécuter conditionnellement une partie du Makefile en fonction de l'existence d'une variable, de sa valeur, etc.

Supposons, par exemple, que nous souhaitions compiler notre projet tantôt en mode debug, tantôt en mode release sans avoir à modifier plusieurs lignes du Makefile pour passer d'un mode à l'autre. Il suffit de créer une variable DEBUG et tester sa valeur pour changer de mode :

### Makefile

```
DEBUG=yes
CC=gcc
ifeq ($(DEBUG),yes)
    CFLAGS=-W -Wall -ansi -pedantic -g
    LDFLAGS=
else
    CFLAGS=-W -Wall -ansi -pedantic
    LDFLAGS=
endif
EXEC=hello
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

all: $(EXEC)
ifeq ($(DEBUG),yes)
    @echo "Génération en mode debug"
else
    @echo "Génération en mode release"
endif

hello: $(OBJ)
    @$(CC) -o $@ $^ $(LDFLAGS)

main.o: hello.h

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

.PHONY: clean mrproper

clean:
    @rm -rf *.o

mrproper: clean
    @rm -rf $(EXEC)
```

Dans ce cas l'exécutable est généré en mode debug, il suffit de changer la valeur de la variable DEBUG pour revenir en mode release.

## 10 - sous-Makefiles

Il arrive parfois que l'on souhaite créer plusieurs Makefiles correspondant à des parties distinctes d'un projet (par exemple : client/serveur, bibliothèques de fonctions, sources réparties dans plusieurs répertoires, etc.).

Toutefois il est souhaitable que ces Makefiles soient appelés par un unique Makefile 'maître' et que les options soient identiques d'un Makefile à l'autre.

Il est ainsi possible d'appeler un Makefile depuis un autre Makefile grâce à la variable \$(MAKE) et de fournir à ce second Makefile des variables définies dans le premier en exportant celles-ci via l'instruction export, avant d'invoquer le second Makefile.

Voyons cela à travers notre petit projet, nous supposons qu'il se situe dans le sous-répertoire hello et que le Makefile maître, situé dans le répertoire principal, définira le compilateur et les options utilisées, nous obtenons alors :

### Makefile maître

```
export CC=gcc
export CFLAGS=-W -Wall -ansi -pedantic
export LDFLAGS=
HELLO_DIR=hello
EXEC=$(HELLO_DIR)/hello

all: $(EXEC)

$(EXEC):
    @(cd $(HELLO_DIR) && $(MAKE))

.PHONY: clean mrproper $(EXEC)

clean:
    @(cd $(HELLO_DIR) && $(MAKE) $@)

mrproper: clean
    @(cd $(HELLO_DIR) && $(MAKE) $@)
```

### Makefile

```
EXEC=hello
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

all: $(EXEC)

hello: $(OBJ)
    @$ (CC) -o $@ $^ $(LDFLAGS)

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

.PHONY: clean mrproper

clean:
    @rm -rf *.o

mrproper: clean
    @rm -rf $(EXEC)
```

## 11 - Bibliographie et liens

- 1 **GNU make**
- 2 Le **manuel** de GNU Make

