Créer et exécuter un Makefile

Informations

Contact : dimitri.boudier@ensicaen.fr

Sources

Tutoriel en français

Tutoriel en anglais

Make manual

Qu'est-ce qu'un Makefile ?

Un Makefile est un fichier dans lequel sont regroupées toutes les commandes de compilation pour un projet logiciel. L'idée est de lancer toutes les compilations nécessaires avec une seule commande dans la console : `make`.

On utilise généralement un Makefile quand le projet prend de l'ampleur (en nombre de fichiers) afin de :

- simplifier le processus de compilation pour le développeur
- faire de la compilation sélective : seuls les fichiers qui doivent être mis à jour sont recompilés, les autres sont conservés (gain de temps de compilation).

Le fichier Makefile doit s'appeler `Makefile` (une majuscule, pas d'extension). On le place généralement à la racine du projet. Pour compiler un projet à partir d'un Makefile, il suffit d'utiliser la console, de se placer dans le même répertoire que le fichier Makefile et de taper `make` dans la console.

1. Makefile minimaliste

1.1. Structure d'un Makefile

Un Makefile se base sur l'utilisation de **règles**. En voici un exemple :

```
mon_exec: mon_fichier.c                #cible: pré-requis
    gcc mon_fichier.c -o mon_exec      #   commande
```

Une **règle (*rule*)** contient généralement trois champs :

- la **cible (*target*)** : ce qui précède les deux points (`:`). Ce sont les fichiers à générer.
- les **pré-requis (*prerequisites*)** : ce qui suit les deux points. Ce sont les fichiers à comparer à la cible (lesquels sont les plus récents ?)
- la **commande (*command*)** : la ligne du dessous; toujours précédée d'une tabulation (pas d'espaces).

1.2. Exécution

En exécutant la commande `make` dans la console, la première règle rencontrée sera évaluée :

- soit la cible est plus récente que les pré-requis et il ne se passe rien
- soit la cible est plus vieille qu'au moins un pré-requis, et la commande sera exécutée (pour mettre à jour la cible)

Dans l'exemple ci-dessous, la première fois que la commande `make` est exécutée, `mon_exec` n'existe pas et la commande est exécutée, créant ainsi le fichier `mon_exec`. Dans un second temps, l'appel à `make` n'entraîne pas l'exécution de la commande car `mon_exec` est plus récent que `mon_fichier.c` (donc pas besoin de mise à jour).

> dboudier:work$ make
> gcc mon_fichier.c -o mon_exec
>
> dboudier:work$ make
> make: 'mon_exec' is up to date.

2. Gérer la compilation d'un projet

2.1. Chaîne de compilation

C'est sûrement transparent pour vous, mais un fichier C n'est pas *directement* compilé en un fichier exécutable. Le processus réel se rapproche plus de celui-ci : `fichier.c` -> `fichier.o` -> `fichier`, avec dans l'ordre un fichier source (en C), un fichier objet (en binaire) et un fichier exécutable (en binaire). Dans la **chaîne de compilation (*toolchain*)** (`gcc` ou `clang` pour vous), le **compilateur (*compiler*)** génère **un** fichier objet à partir d'**un** fichier source et l'**éditeur de liens (*linker*)** génère **un** exécutable à partir de **tous** les fichiers objets du projet.

Si on s'intéresse à ça, c'est parce que nous allons maintenant **compiler plusieurs fichiers sources (en C) en un seul exécutable**, mais pour ça il faut d'abord générer autant de fichiers objets (`.o`, en binaire) que de fichiers sources.

2.2. Makefile avec plusieurs règles

Voici un Makefile répondant au besoin de compiler plusieurs fichiers sources en un seul exécutable.

```
mon_exec: main.o fichier2.o
    gcc main.o fichier2.o -o mon_exec   # Edition des liens
                                        # = (.o + .o + .o ... -> exec)

main.o: main.c fichier2.h
    gcc main.c -c -o main.o             # Compilation = (.c -> .o)

fichier2.o: fichier2.c fichier2.h
    gcc fichier2.c -c -o fichier2.o     # Compilation = (.c -> .o)
```

Regardons ce fichier de bas en haut (de la dernière règle à la première).

La troisième règle est la plus simple : si le fichier objet `fichier2.o` est plus vieux que les fichiers source `fichier2.c` ou header `ficher2.h` correspondants, alors on regénère le fichier objet (mise à jour). Notez l'option `-c` ajoutée à la commande. On indique à gcc "tu compiles seulement", donc pas d'édition des liens (pas de *linker*) et donc le fichier généré est un fichier objet et non un ficher exécutable.

La deuxième l'est presque autant : si le fichier objet `main.o` est plus vieux que le fichier source `main.c` ou que `fichier2.h`, alors on regénère le fichier objet (mise à jour). Mais pourquoi s'intéresser à `ficher2.h` ? Parce que `main.c` a besoin des informations contenues dans le header (il y a un `#include "fichier2.h"` dans le `main.c`). Donc si le header a été modifié, alors il faudra remettre à jour le fichier objet `main.o`.

Enfin, la première règle construit le fichier exécutable `mon_exec` à partir des deux fichiers objets `main.o` et `fichier2.o` (si ces derniers sont plus récents). Notez que cette fois-ci, l'option `-c` a disparu de la commande : on ne s'arrête pas juste après la compilation, mais on va jusqu'au bout du processus pour générer l'exécutable.

Dernière remarque : pourquoi avoir expliqué le fichier de bas en haut et ne pas simplement avoir écrit les règles dans l'ordre inverse ? Parce que quand on tape la commande `make`, c'est la première règle qui est évaluée :

1. `mon_exec` est-il plus vieux que `main.o` ou `fichier2.o` ? Avant d'y répondre, on évalue les règles `main.o` et `fichier2.o`
2. `main.o` est-il plus vieux que `main.c` ou `fichier2.h` ? Si oui, recompilation de `main.o`. Sinon, on passe à la suite.
3. `fichier2.o` est-il plus vieux que `fichier2.c` ou `fichier2.h` ? Si oui, recompilation de `fichier2.o`. Sinon, on passe à la suite.
4. On répond à la première question. Si oui, recompilation de `mon_exec`. Sinon, fin.

```
dboudier:work$ make
gcc main.c -c -o main.o
gcc fichier2.c -c -o fichier2.o
gcc main.o fichier2.o -o mon_exec
```

2.3. Règle de nettoyage

Les règles (et plus précisément les commandes) ne sont pas uniquement dédiées à la compilation. Par exemple on peut ajouter la règle `clean` (souvent présente par habitude).

```
mon_exec: main.o fichier2.o
    gcc main.o fichier2.o -o mon_exec

main.o: main.c fichier2.h
    gcc main.c -c -o main.o

fichier2.o: fichier2.c fichier2.h
    gcc fichier2.c -c -o fichier2.o

clean:
    rm -f *.o
```

La commande permet de supprimer (`rm` = *remove*) tous les fichiers finissant par `.o`. Notons que cette règle n'a pas de dépendance : la commande sera forcément exécutée dès que la règle sera évaluée.

Pour évaluer une règle en particulier, on peut taper son nom dans la console (ici `make clean`).

```
dboudier:work$ ls
fichier2.c  fichier2.h  fichier2.o  main.c  main.o  Makefile  mon_exec

dboudier:work$ make clean
rm -f *.o

dboudier:work$ ls
fichier2.c  fichier2.h  main.c  Makefile  mon_exec
```

Note : la commande `ls` permet d'afficher (*list*) le contenu du répertoire courant.

```
L'ensemble de ce qui a été énoncé jusqu'à présent devrait suffire pour la suite
des TP. Néanmoins la suite vous montre comment faire des Makefile encore plus
évolués.
```

3. Utilisation avancée

Vous entrez maintenant dans la partie avancée du tuto.

En bref, nous parlons ici de rendre le Makefile à la fois plus léger à l'écriture, mais aussi de le rendre modulable de sorte à modifier facilement les options de compilation. Pour les projets de plus grande envergure, la gestion d'un projet structuré selon une arborescence est aussi vu.

3.1. Variables automatiques

On remarque dans le Makefile ci-dessous que les fichiers sont nommés explicitement un bon nombre de fois (alors qu'il n'y a que trois fichiers de base dans le projet).

```
mon_exec: main.o fichier2.o
    gcc main.o fichier2.o -o mon_exec

main.o: main.c fichier2.h
    gcc main.c -c -o main.o

fichier2.o: fichier2.c fichier2.h
    gcc fichier2.c -c -o fichier2.o
```

Un développeur étant souvent faignant, des variables automatiques on été créées pour réduire la dose d'écriture (au détriment de la lisibilité pour les non-initiés). Voici les plus courants :

- `$@` : le nom de la cible
- `$<` : le nom de la première dépendance
- `$^` : le nom de toutes les dépendances

Le Makefile précédent devient alors :

```
mon_exec: main.o fichier2.o
    gcc $^ -o $@            # Compile toutes les dépendances pour créer la cible

main.o: main.c fichier2.h
    gcc $< -c -o $@         # Compile la première dépendance pour créer la cible

fichier2.o: fichier2.c fichier2.h
    gcc $< -c -o $@         # Compile la première dépendance pour créer la cible
```

3.2. Variables utilisateur

L'utilisateur (développeur) peut aussi créer ses variables. Par convention, quatre variables sont habituellement utilisées pour stocker le nom de la chaîne de compilation (`gcc`), les options passées au compilateur, les options passées à l'éditeur de liens et la liste des exécutables. L'initialisation se fait avec l'opérateur `=`, l'utilisation des variables avec la notation `$( )`.

```
CC=gcc                  # Choix de la chaîne de compilation
CFLAGS=-c -W -Wall      # Options pour le compilateur (.c -> .o)
LDFLAGS=                # Options pour le linker (.o + .o + ... -> exec)
EXEC=mon_exec           # Nom de l'exécutable

$(EXEC): main.o fichier2.o
    $(CC) $^ $(LDFLAGS) -o $@

main.o: main.c fichier2.h
    $(CC) $< $(CFLAGS) -o $@

fichier2.o: fichier2.c fichier2.h
    $(CC) $< $(CFLAGS) -o $@

clean:
    rm -f *.o
```

3.3. Génération automatique de la liste des fichiers objet

Il est possible de générer automatiquement la liste des fichiers objets dans la règle `$(EXEC)`, en allant d'abord chercher automatiquement la liste des fichiers sources. On fait cela grâce à deux variables `SRC` et `OBJ` :

```
CC=gcc
CFLAGS=-c -W -Wall
LDFLAGS=

SRC_FILES= $(wildcard *.c)      # Récupère la liste des fichiers terminant par .c
OBJ_FILES= $(SRC_FILES:.c=.o)   # Dans la liste, remplace les ".c" en ".o"
EXEC=mon_exec

$(EXEC): $(OBJ_FILES)           # L'exécutable dépend de la liste des fichiers .o
    $(CC) $^ $(LDFLAGS) -o $@

main.o: main.c fichier2.h
    $(CC) $< $(CFLAGS) -o $@

fichier2.o: fichier2.c fichier2.h
    $(CC) $< $(CFLAGS) -o $@

clean:
    rm -f *.o
```

3.4. Règle générique

Nous savons que chaque fichier source en C doit donner naissance à un fichier objet correspondant. Toujours dans l'optique d'alléger la notation du Makefile, il est possible de définir une règle générique réalisant cette tâche.

```
%.o: %.c                        # Cible = .o ; pré-requis = équivalent .c
    $(CC) $< $(CFLAGS) -o $@    # Compilation (.c -> .o)
```

Néanmoins, il faut garder à l'esprit que cette règle ne fait que comparer un fichier objet `.o` à son fichier source `.c` correspondant, alors qu'il peut aussi avoir des dépendances envers certains headers. Pour cela, on peut écrire une règle spécifique à chaque cible, tout en passant aussi à travers la règle générique :

```
CC=gcc
CFLAGS=-c -W -Wall
LDFLAGS=

SRC_FILES= $(wildcard *.c)
OBJ_FILES= $(SRC_FILES:.c=.o)
EXEC=mon_exec

$(EXEC): $(OBJ_FILES)
    $(CC) $^ $(LDFLAGS) -o $@

main.o: fichier2.h          # Si main.c fait un #include "fichier2.h"

fichier2.o: fichier2.h      # Si fichier2.c fait un #include "fichier2.h"

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

clean:
    rm -f *.o
```

3.5. Arborescence de projet

Rien qu'avec deux fichiers sources et un header, nous avons créé deux fichiers objets et un exécutable, sans compter le Makefile. Avec plus de fichiers sources, le contenu du répertoire de travail deviendrait rapidement brouillon. Pour éviter cela, les projets sont construits autour d'une arborescence structurée et claire. Voici un exemple minimaliste :

- `projet/`, le répertoire du projet
  - `Makefile`, le Makefile se trouve directement à la racine du projet
  - `src/`, un sous-répertoire contenant les fichiers sources (`.c`)
  - `inc/`, un sous-répertoire contenant les fichiers headers (`.h`)
  - `obj/`, un sous-répertoire contenant les fichiers objets (`.o`)
  - `bin/`, un sous-répertoire contenant les fichiers exécutables (`.bin` ou sans extension)

Il faut donc mettre le Makefile à jour de sorte à ce que les chemins d'accès aux fichiers soient gérés. Pour cela on utilise 4 nouvelles variables (nommées respectivement `SRC_DIR`, `INC_DIR`, `OBJ_DIR` et `BIN_DIR`). Ensuite il "suffit" placer les variables aux endroits pertinents, notés `#ici`.

```
CC=gcc
CFLAGS=-c -W -Wall -I./$(INC_DIR)   # Note 2
LDFLAGS=

SRC_DIR=src/
INC_DIR=inc/
OBJ_DIR=obj/
BIN_DIR=bin/

SRC_FILES= $(wildcard $(SRC_DIR)*.c)    #ici
OBJ_FILES= $(subst $(SRC_DIR),$(OBJ_DIR), $(SRC_FILES:.c=.o))   # Note 1
EXEC=$(BIN_DIR)mon_exec     #ici

$(EXEC): $(OBJ_FILES)
    $(CC) $^ $(LDFLAGS) -o $@

${OBJ_DIR}main.o: $(INC_DIR)fichier2.h      #ici

${OBJ_DIR}fichier2.o: $(INC_DIR)fichier2.h  #ici

${OBJ_DIR}%.o: $(SRC_DIR)%.c        #ici
    $(CC) $< $(CFLAGS) -o $@        #ici

clean:
    rm -f ${OBJ_DIR}*.o     #ici

cleanall: clean             # Note 3
    rm -f ${BIN_DIR}*       #ici
```

Note 1 : Pour rappel, la variable `SRC_FILES` contient la liste des noms *complets* des fichiers C, et la commande `$(SRC_FILES:.c=.o)` permet de remplacer l'extension `.c` en `.o` dans le nom de ces fichiers. Mais ce n'est pas suffisant, car la variable `SRC_FILES` contient le nom *complet* des fichiers sources, ce qui inclut le nom du sous-répertoire dans lequel ils sont stockés (`src/` dans cet exemple). Les fichiers objets étant dans un autre sous-répertoire (`obj/` dans cet exemple), la commande `subst $(SRC_DIR),$(OBJ_DIR)` permet de substituer (remplacer) le nom du sous-répertoire `SRC_DIR` par celui du sous-répertoire `OBJ_DIR`. Ainsi la variable `OBJ_FILES` contient la liste des fichiers objets sous la forme `obj/___.o` (le bon sous-répertoire, la bonne extension).

Note 2 : L'option `-I./$(INC_DIR)` a été ajoutée à la variables `CFLAGS` (pour les options du compilateur). Cela permet de signaler à GCC où chercher les headers lorsqu'il rencontre des commandes `#include "___.h"`.

Note 3 : Rien à voir avec l'arborescence du projet, mais une règle a été ajoutée. Elle permet de supprimer les fichiers objets (en appelant la règle `clean`) et de supprimer les exécutables générés. Pour l'appeler, taper `make cleanall` dans la console.