

Fichier « makefile »

Définition

Un fichier « makefile » permet de construire des fichiers objets et des bibliothèques à partir de programmes source et de générer si c'est nécessaire des exécutables. Il peut être utilisé aussi pour réaliser d'autres actions comme l'archivage des données, la remise des travaux etc.

L'intérêt d'un « makefile »

L'instruction suivante sert à compiler 99 fichiers ...

```
g++ -o FichExe Fic1.cpp Fic2.cpp Fic3.cpp ..... Fic99.cpp
```

Si vous modifiez un des fichiers, et vous lancez la commande de nouveau, le compilateur va devoir malgré tout recompiler tous les programmes source de « Fic1.cpp » à « Fic99.cpp », ce qui constitue une perte de temps!

Un fichier « makefile » sait comment identifier le programme qui a été modifié et qui nécessite donc une nouvelle compilation.

Le contenu d'un « makefile »

Un fichier « makefile » contient une liste de cibles, de dépendances et de commandes. Les éléments sont organisés comme suit :

```
cible: dependance  
<tab>commandes
```

La commande « make » exécute les commandes définies par le « makefile » pour mettre à jour une ou plusieurs cibles.

- **La cible :** elle peut représenter le nom d'un fichier ou un objectif (une autre cible) à réaliser (comme remettre les travaux).

- **Les dépendances :** elles représentent les prérequis pour que la cible soit exécutée.

```
Test.o: Test.cpp Test.h
```

La cible est représentée dans cet exemple par le fichier objet « Test.o ». Pour que cette cible soit mise à jour, elle a besoin de deux fichiers d'où le terme dépendance. Dans cet exemple, il y a deux dépendances : les 2 fichiers « Test.cpp » et « Test.h ».

- **Les commandes** : ce sont les actions à exécuter pour mettre à jour la cible.

```
g++ -o Test.exe Test.cpp
```

La commande dans cet exemple est un appel au compilateur « g++ » afin de générer l'exécutable « Test.exe » à partir du code source « Test.cpp ». Les commandes sont toujours précédées d'une tabulation.

Exemple d'un fichier « makefile »

```
all: XTotal

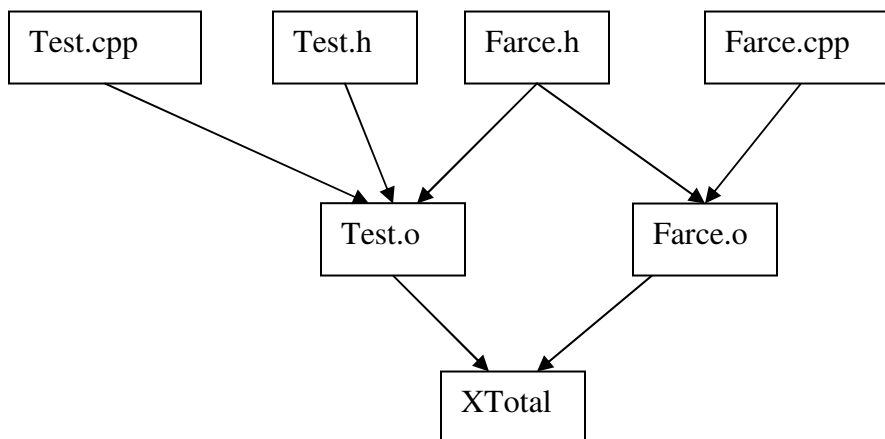
Farce.o: Farce.cpp Farce.h
<tab> g++ -pedantic -Wall -c Farce.cpp -o Farce.o

Test.o: Test.cpp Test.h Farce.h
<tab> g++ -pedantic -Wall -c Test.cpp -o Test.o

XTotal: Farce.o Test.o
<tab> g++ -o XTotal Farce.o Test.o

clean:
<tab> rm *.o XTotal
```

Le graphe de dépendance pour un tel fichier est comme suit :



Dans cet exemple, nous avons défini 4 cibles : « all », « Farce.o », « Test.o » et « XTotal ».

La cible « all » représente la cible globale permettant de regrouper toutes les cibles qui ont besoin d'être mises à jour. Dans cet exemple, la cible « all » n'est qu'un alias à la cible « XTotal ».

```
make
```

En lançant la commande « make » sans arguments, cette dernière va chercher dans le répertoire courant un fichier portant le nom « makefile ». S'il existe, il sera exécuté sinon il y aura un message d'erreur.

Si le fichier existe, la cible « all » sera exécutée en premier si elle existe dans le fichier « makefile ». Dans le cas contraire la première cible trouvée dans le fichier « makefile » sera exécutée en premier.

Il possible aussi de préciser une cible sur la ligne de commande comme par exemple :

```
make all
```

Pour construire la cible « all », la commande « make » a besoin de regarder l'état des prérequis. Si ce sont des cibles, la commande « make » doit en premier lieu les construire.

Ainsi donc pour construire « all », la commande « make » a besoin de construire d'abord la cible « Farce.o » puis par la suite la cible « Test.o ».

Pour construire la cible « Farce.o », la commande « make » doit tenir compte de 2 prérequis : les fichiers « Farce.cpp » et « Farce.h ».

De la même manière, pour construire la cible « Test.o », la commande « make » doit tenir compte de 3 prérequis : les fichiers « Test.cpp », « Test.h » et « Farce.h ».

Finalement, pour construire la cible « XTotal », la commande « make » doit tenir compte des 2 prérequis qui sont des cibles « Farce.o » et « Test.o ».

À noter les points suivants :

1- Toute modification apportée sur le fichier « Farce.h » va forcer la mise à jour des cibles « Farce.o », « Test.o » et « XTotal » car si nous examinons attentivement les dépendances « Farce.o » et « Test.o », celles-ci dépendent de « Farce.h » et que « XTotal » dépend de « Farce.o » et « Test.o ».

2- Par contre si uniquement le fichier « Test.h » est modifié, les cibles « Test.o » et « XTotal » seront de facto mises à jour. En effet, « Test.o » dépend de « Test.h » et « XTotal » dépend de « Test.o ».

La commande « make » ne va donc pas réaliser l'opération de mise à jour que si un des prérequis est plus récent que la cible elle même. Ce mécanisme permet à la commande « make » de ne compiler que ce qui est nécessaire et pas plus.

Si après avoir exécuté une première fois la commande « make », vous exécutez par la suite la commande « make all » sans avoir préalablement modifié une des dépendances, la commande « make » va afficher en sortie le message suivant :

```
make: Nothing to be done for `all'.
```

3- Si vous exécutez la commande :

```
make Farce.o
```

Seuls la cible « Farce.o » sera mise à jour si ses dépendances sont plus récentes qu'elle.

4- Si vous exécutez la commande :

```
make clean
```

Cette dernière va exécuter la cible « clean »

```
clean:  
<tab>rm *.o XTotal
```

La cible « clean » n'a pas besoin de dépendances. Elle va se charger d'exécuter uniquement la commande « rm » permettant d'effacer les fichiers ayant l'extension « .o » et l'exécutable « XTotal ».

5- Il est possible de nommer autrement le fichier « makefile ». Dans ce cas, il faudra utiliser l'option « -f » pour préciser le fichier « makefile » à utiliser avec la commande « make ».

```
make -f makefiletp2
```

Dans cet exemple, nous allons utiliser le fichier « makefiletp2 » au lieu du fichier par défaut « makefile ».

Définition de variables

Nous pouvons définir une série de variables dans un fichier « makefile » en utilisant la syntaxe suivante :

```
NOM_VARIABLE = VALEUR
```

Par exemple :

```
CXX = g++  
CXXFLAGS = -Os -Wall  
OPT = -pedantic
```

Où :

- « **-Os** » : permet optimiser la taille du code.
- « **-Wall** » : affiche tous les avertissements.
- « **-pedantic** » : s'assure que le code respecte strictement les spécifications du langage.

L'appel à la variable se fait en utilisant le symbole « \$ » comme suit :

```
CXX = g++  
CXXFLAGS = -Os -Wall  
OPT = -pedantic  
  
all: XTotal  
  
Farce.o: Farce.cpp Farce.h  
<tab> $(CXX) $(CXXFLAGS) $(OPT) -c Farce.cpp -o Farce.o  
  
Test.o: Test.cpp Test.h Farce.h  
<tab> $(CXX) $(CXXFLAGS) $(OPT) -c Test.cpp -o Test.o  
  
XTotal: Farce.o Test.o  
<tab> $(CXX) -o XTotal Farce.o Test.o
```

En réalité les variables « CXX » et « CXXFLAGS » sont déjà définies implicitement et elles sont associées au langage « C++ ». Le fait de les définir dans le fichier « makefile » équivaut à une redéfinition. Pour le langage « C », nous utilisons plutôt les variables « CC » et « CFLAGS ».

Définir une variable dans un fichier « makefile » va nous permettre de la modifier plus aisément à un seul endroit du fichier.

Variables automatiques

« make » permet d'utiliser une série de variables automatiques comme suit :

<code>\$@</code>	Le nom de la cible associée à la règle
<code>\$<</code>	Le nom de la première dépendance
<code>\$?</code>	La liste de toutes les dépendances (séparées par un espace blanc) qui sont plus récentes que la cible
<code>^</code>	La liste de toutes les dépendances (séparées par un espace blanc)
<code>\$*</code>	Le nom du fichier sans suffixe

Ainsi pour cet exemple :

```
Farce.o: Farce.cpp Farce.h
<tab> $(CXX) $(CXXFLAGS) $(OPT) -c Farce.cpp -o Farce.o
```

Les variables automatiques associées seront définies comme suit :

<code>\$@</code>	Farce.o
<code>\$<</code>	Farce.cpp
<code>\$?</code>	Farce.cpp Farce.h
<code>^</code>	Farce.cpp Farce.h
<code>\$*</code>	Farce

De ce fait, une écriture équivalente :

```
Farce.o: Farce.cpp Farce.h
<tab> $(CXX) $(CXXFLAGS) $(OPT) -c $< -o $@
```

Pour l'exemple suivant :

```
XTotal: Farce.o Test.o
<tab> $(CXX) -o XTotal Farce.o Test.o
```

Les variables automatiques associées sont :

<code>\$@</code>	XTotal
<code>^</code>	Farce.o Farce.o

Une écriture équivalente :

```
XTotal: Farce.o Test.o
<tab> $(CXX) $^ -o $@
```

Ainsi le fichier « makefile » devient comme suit :

```
CXX = g++
CXXFLAGS = -Os -Wall
OPT = -pedantic

objets = Farce.o Test.o
programme = XTotal

all: $(programme)

Farce.o: Farce.cpp Farce.h
<tab> $(CXX) $(CXXFLAGS) $(OPT) -c $< -o $@

Test.o: Test.cpp Test.h Farce.h
<tab> $(CXX) $(CXXFLAGS) $(OPT) -c $< -o $@

$(programme): $(objets)
<tab> $(CXX) -o $@ $^
```

Les règles génériques

Nous pouvons définir dans un fichier « makefile » des règles dites génériques qui vont agir en fonction de l'extension des fichiers traités et non pas leurs noms.

Ainsi donc l'instruction suivante :

```
%.o : %.cpp
<tab> $(CXX) $(CXXFLAGS) $(OPT) -c $< -o $@
```

Elle permet de générer une cible pour chaque fichier source. Le symbole « % » remplace le nom du fichier. Le nom du fichier n'est donc pas important à ce stade.

Le fichier « makefile » complet est comme suit :

```
# variables automatiques
CXX = g++
CXXFLAGS = -Os -Wall
OPT = -pedantic
programme = XTotal
objets = Farce.o Test.o

# cible globale
all: $(programme)

# les autres cibles, sous le format:

# cible: dependance
# commandes

# Attention, tabulation nécessaire avant chaque commande

%.o : %.cpp
    $(CXX) $(CXXFLAGS) $(OPT) -c $< -o $@

$(objets): Farce.h

$(programme): $(objets)
    $(CXX) -o $@ $^

clean:
    rm Farce.o Test.o $(programme)
```

Si vous décidez d'ajouter un fichier objet supplémentaire, il n'est pas nécessaire d'ajouter une règle dans le fichier « makefile ». Il suffit en effet d'ajouter le nom de fichier objet dans la ligne :

```
objets = Farce.o Test.o
```

À noter qu'il était nécessaire de préciser les dépendances pour la variable « objet ». Lors de la compilation de « Farce.cpp » elle va faire appel de facto au fichier « Farce.h ». Il en est de même pour « Test.cpp » qui fera appel à « Test.h ». Cependant « Test.cpp » a besoin aussi de « Farce.h », c'est pour cette raison qu'il fallait l'inclure dans les dépendances de la variable « objet ».

Commandes supplémentaires

1- « make » contient aussi une série de commandes très variées. Nous allons citer juste quelques exemples :

```
ifeq ($(OS), Windows_NT)
programme = XTotal.exe
endif
```

La commande « ifeq » est un « if conditionnel ». Dans cet exemple, « ifeq » compare la valeur de la variable « OS » avec la chaîne « Windows_NT ». Si les deux valeurs sont identiques, alors nous allons modifier la valeur de la variable « programme ». Nous terminons le if conditionnel par l'instruction « endif ». Ce test nous permet de savoir si nous sommes en présence d'un système « Windows ». En effet par défaut le compilateur « g++ », sous « Windows », ajoute l'extension « .exe » au programme exécutable.

```
clean:
<tab>rm Farce.o Test.o $(programme)
```

L'appel à la cible « clean » permet d'effacer les fichiers « Farce.o », « Test.o » et celui associé à « programme ». Sous Windows cette variable prend la valeur « XTotal.exe » alors que sous une autre architecture, elle va valoir « XTotal ». Le test est donc nécessaire pour s'assurer d'effacer le bon fichier.

2- Affichage en sortie :

```
$(programme): $(objets)
<tab>echo "programme: $(programme) ==> objets: $(objets) "
<tab>$(CXX) -o $@ $^
```

La commande « echo » permet d'afficher du texte en sortie lors de l'exécution du fichier « makefile ».

3- Pour faire des commentaires dans un fichier « makefile », nous utilisons le symbole « # » :

```
# Attention, tabulation nécessaire avant chaque commande
```

4- Lien utile : la documentation complète de « gnu make »

<http://www.gnu.org/software/make/>