L'outil MAKE

par Emmanuel Delahaye (Espace personnel d'Emmanuel Delahaye)

Date de publication : 28 janvier 2008

Dernière mise à jour : 25 avril 2009

Cet article présente un exemple concret d'utilisation de l'outil MAKE dans le cadre du développement d'un projet C par une approche didactique et pragmatique.

Votre avis et vos suggestions sur cet article nous intéressent !
Alors après votre lecture, n'hésitez pas :

I - Introduction

La production de code (compilation, édition de liens) est assez fastidieuse et difficile à gérer 'à la main' (ligne de commande, scripts...) dès que le projet devient conséquent et évolutif (en fait dès qu'il y a plus d'un fichier en jeu).

Il existe differents outils qui peuvent aider à la gestion des projets. le plus simple et le plus connu est sans aucun doute make (ou une de ses variantes) qui est basé sur une certain nombre de principes et de syntaxes simples.

Dans ce document, je m'efforce de n'utiliser que les caractéristiques les plus courantes qui, bien que l'outil ne soit pas normalisé, devraient se retrouver sur la plupart des versions de make (testé avec le make de Borland et le make de GCC sous DOS/Windows).

Ce document ne prétend en aucun cas être exhaustif. Il s'agit de montrer un exemple concret d'utilisation dans le cadre du développement d'un projet C. L'approche reste didactique et pragmatique.

II - Principes

La structure de base est la suivante

```
<cible> : <dependance[s]>
-TAB-&lt;action[s]>
```

Le fichier qui contient les commandes de l'outil MAKE s'appelle 'makefile' ou 'Makefile' par défaut. L'invocation de 'make' exécute ce fichier si il se trouve sur le répertoire courant.

```
> make
```

On peut exécuter un autre fichier de commandes, mais il faut alors utiliser le paramètre -f

```
> make -f projet.mak
```

La cible sera réalisée en fonction des dépendances et au moyen des actions.

Il y a une tabulation avant chaque action.

La cible par défaut est la première de la liste. On mettra donc la cible finale en premier et les moyens d'y arriver à la suite.

Les caractères situés après un # sont ignorés (commentaires)

L'invocation se fait par

```
> make
```

Cible par défaut

```
> make <cible>
```

Cible particulière

III - Exemple de base

Soit la trilogie classique permettant de tester un petit module

```c
/* main.c */
#include "inc/f.h"

int main (void)
{
   f ();
   return 0;
}
```

```c
/* f.c */
#include "inc/f.h"
#include <stdio.h>

void f (void)
{
   puts ("hello world");
}
```

```c
#ifndef H_F
#define H_F

/* f.h */

void f (void);

#endif
```

le projet est organisé ainsi

```
+/<projet>
 |
 +-+/inc
 | |
 | +-f.h
 |
 +-+/src
 | |
 | +-f.c
 |
 +-+/tu
   |
   +-+/src
   |   |
   |   +-main.c
   |
   +-+-/out
   | |
   | +-f.o
   | |
   | +-main.o
   | |
   | +-tu.exe
   |
   +-Makefile
```

Voici une version basique mais commentée du fichier Makefile servant à réaliser le mini projet.

```
# Makefile

# Cible par defaut (make)
# Cible            : executable
# Dependances      : les fichiers objets
```

```
# Actions          : edition de lien

out/tu.exe: out/main.o out/f.o
   gcc out/main.o out/f.o -o out/tu.exe

# Cible            : fichier objet
# Dependances      : le fichier source et les fichiers inclus
# Actions          : compilation

out/main.o : src/main.c ../inc/f.h
   gcc -c -I /mesdoc~1/emmanuel/site/make src/main.c -o out/main.o

out/f.o    : ../src/f.c ../inc/f.h
   gcc -c -I /mesdoc~1/emmanuel/site/make ../src/f.c -o out/f.o
```

IV - Ajout de cibles

Il est possible d'ajouter des cibles supplémentaires permettant une action particulière. Par exemple, l'effacement de tous les fichiers produits, afin de repartir sur des bases 'saines'.

Une cible sans dépendance sera invoquée systématiquement.

Une cible 'supplémentaire' doit être placée après la cible 'par défaut'.

Le Makefile est donc modifié comme ceci :

```
# Makefile

out/tu.exe: out/main.o out/f.o
   gcc out/main.o out/f.o -o out/tu.exe

out/main.o : src/main.c ../inc/f.h
   gcc -c -I /mesdoc~1/emmanuel/site/make src/main.c -o out/main.o

out/f.o    : ../src/f.c ../inc/f.h
   gcc -c -I /mesdoc~1/emmanuel/site/make ../src/f.c -o out/f.o

# Cible            : fictive (make clean)
# Dependances      :
# Actions          : effacement des fichiers produits dans /out

clean:
   del out/*.o
   del out/tu.exe
```

Pour effacer le projet, il suffit d'invoquer la cible 'clean'

```
> make clean
```

'clean' est un nom de cible courament utilisé mais pas réservé. On aurait pu aussi bien utiliser 'nettoyage'...

V - Utilisation des variables

Le fichier Makefile ci-dessus montre beaucoup de répétitions. Il est possible de simplifier l'écriture et la maintenance de ce fichier en utilisant des 'variables' qui sont en fait plutôt des macros de substitution de texte.

V-A - Définition d'une variable

```
<variable> = <valeur>
```

V-B - Utilisation d'une variable

```
... $(<variable>) ...
```

V-C - Amélioration de l'exemple

Les parties répétitives sont 'factorisées' avec des variables. D'autre part, certains éléments pouvant dépendre de la plateforme de développement font aussi l'objet d'une abstraction (compilateur, paramètres etc.).

```
# <projet>/tu/Makefile

# repertoire projet
DPROJ = /mesdoc~1/emmanuel/site/make

# repertoire de sortie
DOUT = out

# nom de l'executable (cible par defaut)
BIN = $(DOUT)/tu.exe


# liste des objets (cibles intermediaires)
OBJS = $(DOUT)/main.o $(DOUT)/f.o

# chemin des fichiers inclus pour le compilateur
INCLUDES = -I  $(DPROJ)

# liste des dependances autres que le source
DEPS = ../inc/f.h

# commandes de compilation et d'edition de lien
CC = gcc
LD = gcc

# Commandes simplifiees par l'utilisation des variables.

$(BIN): $(OBJS)
   $(LD) $(OBJS) -o $(BIN)

out/main.o : src/main.c  $(DEPS)
   $(CC) -c $(INCLUDES) src/main.c -o out/main.o

out/f.o    : ../src/f.c  $(DEPS)
   $(CC) -c $(INCLUDES) ../src/f.c -o out/f.o

clean:
   del $(OBJS)
   del $(BIN)
```

Il devient maintenant extrêmement simple d'ajouter des paramètres pour le compilateur

```
# <projet>/tu/Makefile
```

```
DPROJ = /mesdoc~1/emmanuel/site/make
DOUT = out
BIN = $(DOUT)/tu.exe
OBJS = $(DOUT)/main.o $(DOUT)/f.o
INCLUDES = -I  $(DPROJ)
DEPS = ../inc/f.h
CC = gcc
LD = gcc
CFLAGS = -W -Wall -O1 $(INCLUDES)

$(BIN): $(OBJS)
   $(LD) $(OBJS) -o $(BIN)

out/main.o : src/main.c $(DEPS)
   $(CC) $(CFLAGS) -c src/main.c -o out/main.o

out/f.o    : ../src/f.c $(DEPS)
   $(CC) $(CFLAGS)  -c ../src/f.c -o out/f.o

clean:
   del $(OBJS)
   del $(BIN)
```

VI - Variables automatiques

Il existe d'autres variables prédéfinies ou automatiques permettant de simplifier encore le Makefile. Voici une liste indicative :

$@	Nom de la cible
$^	Liste des dépendances
$<	Nom de la première dépendance

Attention, le résultat n'est pas garanti avec tous les outils make. Voici les sorties générées par différents outils make à partir de ce fichier de test:

```
# <projet>/Makefile
# Test des variables automatiques

cible.x: dep1.a dep2.b dep3.c
   @echo "@" : $@
   @echo "^" : $^
   @echo "<" : $<

dep1.a :
   @echo $@ > $@

dep2.b :
   @echo $@ > $@

dep3.c :
   @echo $@ > $@

clean:
   del dep*.*
```

```
GNU Make 3.80
Copyright (C) 2002  Free Software Foundation, Inc.

"@" : cible.x
"^" : dep1.a dep2.b dep3.c
"<" : dep1.a
```

```
MAKE Version 3.6  Copyright (c) 1992 Borland International

Available memory 131387392 bytes

"@" : cible.x
"^" : $^
"<" : cible.x
```

Si on utilise GNU make, on peut simplifer le Makefile comme ceci:

```
# <projet>/tu/Makefile

DPROJ = /mesdoc~1/emmanuel/site/make
DOUT = out
BIN = $(DOUT)/tu.exe
OBJS = $(DOUT)/main.o $(DOUT)/f.o
DEPS = ../inc/f.h
INCLUDES = -I  $(DPROJ)
CC = gcc
LD = gcc
CFLAGS = -W -Wall -O1 $(INCLUDES)
```

```
$(BIN): $(OBJS)
   $(LD) $(OBJS) -o $@

out/main.o : src/main.c $(DEPS)
   $(CC) $(CFLAGS) -c $< -o $@

out/f.o    : ../src/f.c  $(DEPS)
   $(CC) $(CFLAGS)  -c $< -o $@

clean:
   del $(OBJS)
   del $(BIN)
```