Ecrire un Makefile, sans douleur et en quelques leçons.

M. Billaud
Département Informatique
Institut Universitaire de Technologie
Université Bordeaux 1

Mai 2010

Résumé

Ce document montre comment écrire un *Makefile* lisible pour un projet de programmation, en tirant profit des dépendances implicites, règles par défaut etc.

Table des matières

1 A quoi ça sert ?

Un fichier *Makefile* est un texte qui indique à la commande `make`comment fabriquer des fichiers à partir d'autres fichiers.

On rentre dans un fichier `Makefile` des règles qui indiquent comment on fabrique les fichiers, et la commande `make` lance la fabrication.

Le plus souvent, c'est utilisé pour se simplifier la vie pour le développement des programmes d'une certaine taille, dont les sources sont répartis en plusieurs morceaux compilés séparément.

Mais commençons par un exemple simple.

1.1 Exemple : un source C++

Vous avez écrit un programme `hello.cc`

```
#include <iostream>

int main() {
   std::cout << "Hello, world" << std::endl;
}
```

"à la main" vous compilez ce programme en lançant la commande

```
g++ -Wall  -o hello hello.cc
```

1.2 Un premier *Makefile* :

Vous obtiendrez le même résultat en créant un fichier `Makefile` qui contiendrait

```
hello: hello.cc
	g++ -Wall  -o hello hello.cc
```

(attention, la seconde ligne commence par une *tabulation*, pas par des espaces)
et en lançant la commande `make` quand vous voulez recompiler.

1.3 Explications

Ce `Makefile` se lit ainsi

- Le fichier *cible* `hello` dépend de `hello.cc`. On considère qu'il est à jour si il existe (!) et qu'il a été fabriqué ¡i¿après¡/i¿ la dernière modification de `hello.cc`
- pour le (re)-fabriquer, il faut lancer la commande `g++ ...`

1.4 Avantages

Pour recompiler le source

1. plus besoin de relancer une longue commande, il suffit de taper « `make` »
2. si le fichier `hello` est à jour, la commande `make` ne recompilera pas `hello.cc`, puisque c'est inutile.

Sur un exemple aussi simple (un seul source à compiler), l'intérêt d'avoir à apprendre l'utilisation des Makefiles n'est pas flagrant, mais ça va venir.

1.5 Avec plusieurs cibles

Un `Makefile` comporte généralement plusieurs cibles, voyons un exemple plus détaillé

```
hello: hello.cc
	g++ -Wall -o hello hello.cc

hello.pdf: hello.cc Makefile
	a2ps -o - hello.cc Makefile | ps2pdf - hello.pdf

#
# Cibles habituelles
#
clean:
	rm -f *~

mrproper: clean
	rm -f hello
```

Quelques explications

1. La commande « `make` » sans paramètre lance la fabrication (éventuelle) de la première cible `hello`. On aurait le même effet en lançant « `make hello` ».
2. « `make hello.pdf` » fabrique un fichier PDF contenant le source et le makefile.
3. « `make clean` » fait le ménage dans les fichiers intermédiaires (fichiers de sauvegarde l'éditeur de textes, etc)
4. « `make mrproper` » fait le grand ménage, en ne conservant normalement que les sources de l'application.

Recommandation : Avoir ces cibles `clean` et `mrproper` (Monsieur Propre), est une pratique usuelle (et fortement conseillée).

2 Makefiles pour la compilation séparée

C'est la compilation séparée qui fait apparaître tout le charme du `Makefile` aux yeux des programmeurs.

2.1 Un exemple

Vous avez eu l'idée d'écrire une fonction `afficher(string message);` dans un fichier séparé

```
// fichier afficher.cc

#include <iostream>
#include "afficher.h"

void afficher (std::string message)
{
   std::cout << message << std::endl;
}
```

Le prototype est déclaré dans un fichier d'entête :

```
// fichier afficher.h

#include <iostream>

void afficher (std::string message);
```

et le programme `hello.cc` y fait référence

```
// fichier hello.cc

#include "afficher.h"

int main()
{
    afficher("Bonjour, monde");
}
```

2.2 Le Makefile, version naïve

La compilation séparée comportera deux étapes
- la compilation séparée des deux sources (.cc) pour fabriquer les modules objets (.o)
- l'édition des liens des fichiers objets (.o) pour constituer l'exécutable

```
hello: hello.o afficher.o
	g++ -o hello hello.o afficher.o

hello.o: hello.cc afficher.h
	g++ -Wall -c hello.cc

afficher.o: afficher.cc afficher.h
	g++ -Wall -c afficher.cc


hello.pdf: hello.cc afficher.h afficher.cc Makefile
	a2ps -o - hello.cc afficher.h afficher.cc Makefile \
		| ps2pdf - hello.pdf

#
# Cibles habituelles
#
clean:
	rm -f *~ *.o

mrproper: clean
	rm -f hello
```

3 Variables cible, liste de dépendances, première dépendance

Le Makefile ci-dessus peut être largement simplifié en utilisant 3 variables prédéfinies.

`$@`	cible
`$^`	liste des dépendances
`$<`	première dépendance

Dans une commande,

- `$@` contient la *cible* (target). La première règle (édition des liens) pourrait s'écrire

```
hello: hello.o afficher.o
	g++ -o $@ hello.o afficher.o
```

- `$^` contient la *liste des dépendances*. La première règle peut être simplifiée davantage

```
hello: hello.o afficher.o
	g++ -o $@ $^
```

de même que la 3ième

```
hello.pdf: hello.cc afficher.h afficher.cc Makefile
	a2ps -o - $^ | ps2pdf - $@
```

qui devient beaucoup plus lisible.

- `$<`, qui contient la *première dépendance*, rend service pour les compilations séparées

```
hello.o: hello.cc afficher.h
	g++ -Wall -c $<
```

3.1 Makefile avec variables cible, dépendances...

```
hello: hello.o afficher.o
	g++ -o $@ $^

hello.o: hello.cc afficher.h
	g++ -Wall -c $<

afficher.o: afficher.cc afficher.h
	g++ -Wall -c $<

hello.pdf: hello.cc afficher.h afficher.cc Makefile
	a2ps -o - $^ | ps2pdf - $@

# ...
```

4 Variables définies par l'utilisateur

On peut définir des variables pour travailler plus commodément. Par exemple on s'en sert pour définir la liste des fichiers sources, la liste des entêtes, et la liste de modules objet.

```
sources=hello.cc afficher.cc
entetes=afficher.h
objets=$(sources:.cc=.o)

hello: $(objets)
	g++ -o $@ $^

hello.o: hello.cc afficher.h
	g++ -Wall -c $<

afficher.o: afficher.cc afficher.h
	g++ -Wall -c $<
```

```
hello.pdf: $(sources) $(entetes) Makefile
        a2ps -o - $^ | ps2pdf - $@

# ...
```

- l'affectation se passe de commentaires
- on peut ajouter des élements à une variable. On aurait pu écrire :

```
sources   = hello.cc
sources  += afficher.cc
...
```

- l'expansion d'une variable se fait par "dollar parenthèses"
- possibilité de substitution pendant l'expansion. Ligne 3, la liste des modules objets est déduite de la liste des fichiers sources, en remplaçantles suffixes `.cc` par `.o`

5 Les commandes par défaut

La commande `make` possède un stock de dépendances et de règles par défaut qu'on peut mettre à profit pour allèger encore les Makefiles.

Par exemple, si vous avez dans votre répertoire un fichier « `prog.cc` » la commande « `make prog.o` » lance automatiquement la commande

```
g++    prog.cc   -o prog
```

sans qu'on ait écrit quoi que ce soit dans le `Makefile`.

C'est le résultat

- d'une *dépendance implicite* : si `prog.cc` existe dans le répertoire, alors le fichier `prog` dépend de `prog.cc`
- d'une *commande par défaut* : pour fabriquer un exécutable (sans suffixe) à partir d'un source C++ (nom + suffixe .cc), on lance la commande de compilation adaptée `g++ ...` si rien d'autre n'est précisé.

5.1 Utiliser les variables prédéfinies

Ici se pose un petit problème : conserver l'option « `-Wall` » pour les compilations séparées.

Heureusement, les commandes par défaut sont paramétrables par l'intermédiaire de variables. Ici on affecte donc l'option à la variable `CXXFLAGS`

```
CXXFLAGS=-Wall

sources=hello.cc afficher.cc
entetes=afficher.h
objets=$(sources:.cc=.o)

hello: $(objets)
        g++ -o $@ $^

hello.o: hello.cc afficher.h
afficher.o: afficher.cc afficher.h

###

hello.pdf: $(sources) $(entetes) Makefile
        a2ps -o - $^ | ps2pdf - @<
```

```
clean:
	rm -f *~ *.o

mrproper: clean
	rm -f hello
```

5.2 Redéfinir une règle par défaut

Pour la première cible, malheureusement, la commande par défaut ne convient pas. Si on écrivait

```
CXXFLAGS=-Wall

hello: $(objets)

...
```

`make` lancerait « `gcc -o hello hello.o afficher.o` » (au lieu de g++), donc sans faire référence à la bibliothèque standard C++.

Un moyen de s'en sortir est de redéfinir les commandes par défaut, en disant :

> pour fabriquer un fichier exécutable (sans suffixe) à partir de son module objet (suffixe .o) et d'autres éventuellement, il faut lancer la commande d'éditions des liens adaptée à C++.

Ce qui s'écrit :

```
%: %.o
	$(LINK.cc) -o $@ $^
```

Comme vous l'avez deviné, la variable prédéfinie `LINK.cc` contient la commande qui va bien pour l'édition des liens.

5.3 Makefile

```
CXXFLAGS=-Wall

sources=hello.cc afficher.cc
entetes=afficher.h
objets=$(sources:.cc=.o)

%: %.o
	$(LINK.cc) -o $@ $^

hello: $(objets)
hello.o: hello.cc afficher.h
afficher.o: afficher.cc afficher.h

###

hello.pdf: $(sources) $(entetes) Makefile
	a2ps -o - $^ | ps2pdf - $@

clean:
	rm -f *~ *.o
```

```
mrproper: clean
	rm -f hello
```

6 Construction automatique des dépendances

On peut faire encore mieux : s'éviter en grande partie la fastidieuse écriture des dépendances.

6.1 la commande `makedepend`

La commande `makedepend` est un outil complémentaire qui effectue à votre place la recherche des dépendances entre fichiers sources.

En pratique, si on tape

```
makedepend hello.cc afficher.cc
```

les lignes suivantes

```
# DO NOT DELETE

afficher.o: afficher.h
hello.o: afficher.h
```

sont ajoutées à la fin du `Makefile`, remplaçant éventuellement les lignes qui étaient déjà après "DO NOT DELETE".

Ces dépendances sont obtenues en examinant les fichiers cités, pour trouver quels fichiers sont inclus (inclusions à plusieurs niveaux éventuellement)

Les dépendances calculées se combinent harmonieusement avec les dépendances implicites (`afficher.o : afficher.cc`, etc.) et les commandes par défaut pour que tout se passe bien. Il n'y a donc plus besoin d'indiquer comment fabriquer les modules objets. C'est fait pour.

6.2 Makefile final

Voici donc la version finale du Makefile :

```
CXXFLAGS=-Wall

sources = hello.cc afficher.cc
entetes = afficher.h
objets  = $(sources:.cc=.o)

%: %.o
	$(LINK.cc) -o $@ $^

hello: $(objets)

###

hello.pdf: $(sources) $(entetes) Makefile
	a2ps -o - $^ | ps2pdf - $@

clean:
	rm -f *~ *.o *.bak

mrproper :clean
	rm -f hello
```

```
depend:
	makedepend $(sources)
```

Comparez avec la première version...

6.3 Mode d'emploi

1. Avant la première utilisation, faire « `make depend` pour créer la liste des dépendances
2. idem quand vous ajoutez de nouveaux fichiers sources, ou que vous changez les `#include` dans vos programmes.

7 Un exemple plus complexe (Programmation Objet)

Prenons un exemple plus complexe : deux programmes `paie.cc` et `emploiDuTemps.cc` qui font référence à des classes `Titulaire` et `Vacataire` dérivées de `Enseignant` (classe abstraite). Ces mêmes classes sont utilisées dans 2 programmes de test `testTitulaire.cc` et `testVacataire.cc`.

Construction par morceaux : tout bien compté, il y a donc 3 classes avec leurs entêtes

```
classes=Enseignant.cc Titulaire.cc Vacataire.cc
```

et 4 programmes dont il faudra produire les exécutables

```
progs=testTitulaire.cc testVacataire.cc  paie.cc emploiDuTemps.cc
```

avec les dépendance explicites

```
paie:           paie.o           Enseignant.o Titulaire.o Vacataire.o
emploiDuTemps: emploiDuTemps.o  Enseignant.o Titulaire.o Vacataire.o
testTitulaire: testTitulaire.o  Enseignant.o Titulaire.o
testVacataire: testVacataire.o  Enseignant.o Vacataire.o
```

ce qui nous fait 10 sources, 7 modules objets et 4 exécutables.
Mais le makefile reste à taille humaine !

```
CXXFLAGS=-Wall

applis  = paie.cc emploiDuTemps.cc
tests   = testTitulaire.cc testVacataire.cc
classes = Enseignant.cc Titulaire.cc Vacataire.cc

entetes = $(classes:.cc=.h)
execs   = $(tests:.cc) $(progs:.cc=)

all: $(execs)

paie:           paie.o           Enseignant.o Titulaire.o Vacataire.o
emploiDuTemps: emploiDuTemps.o  Enseignant.o Titulaire.o Vacataire.o

testTitulaire: testTitulaire.o  Enseignant.o Titulaire.o
testVacataire: testVacataire.o  Enseignant.o Vacataire.o

listing.pdf: $(applis)  $(entetes) $(classes) Makefile
	a2ps -o - $^ | ps2pdf - $@

listing-tests.pdf: $(tests) $(entetes) $(classes) Makefile
```

```
        a2ps -o - $^ | ps2pdf - @<

#########

%: %.o
        $(LINK.cc) -o $@ $^

clean:
        rm -f *~ *.o *.bak

mrproper :clean
        rm -f $(execs)

depend:
        makedepend $(progs) $(classes)
```