

# PoP C++ Série 1 niveau 0

## Exercice 7 (niveau 0): Un exemple de Makefile

Admettons que l'on ait écrit un programme séparé en modules de la manière suivante :

- gestion d'un ensemble de fruits, dont les fonctions exportées sont déclarées dans **fruits.h** (header) et définies dans le fichier source **fruits.cc**
- gestion d'un ensemble de personnes, avec les fichiers **persons.h**, **persons.cc** qui dépend de **fruits.h**
- Le programme principal (comprenant la fonction main()), dans le fichier **eatFruit.cc** dépend de **fruits.h**, **persons.h**

**Dessinez le graphe des dépendances à partir de ces indications (solution en fin de fichier).**

Un fichier Makefile de base contient une **règle de dépendance** pour *chaque fichier objet* et une règle supplémentaire pour *l'exécutable*. La syntaxe d'une règle fait apparaître pour chaque **cible** l'ensemble des fichiers dont elle dépend ; les fichiers sont séparés par un espace :

**cible** : **dependance1** **dependance2** etc...

Nous avons une règle de dépendance pour chaque module ,c'est à dire **une cible par fichier objet** résultant de la compilation du fichier source .cc, et **une cible supplémentaire pour lier tous les fichiers objets en un programme exécutable**.

Les **dépendances** de chacune de ces cibles sont déterminées par les directives d'inclusion présentes dans les fichiers de déclaration (en-tête .h) et d'implémentation (source .cc).

On ignore les dépendances vers des fichiers que l'on ne peut pas modifier comme par exemple les fichier en-tête de la bibliothèque standard du C++.

En résumé, si une dépendance est modifiée alors sa cible doit être remise à jour. Cela est fait avec une ou plusieurs **commande** qui apparaissent sous la règle de dépendance dans le fichier Makefile. Les commandes de compilation des cibles sont les plus fréquentes mais ce ne sont pas les seules.

**Remarque:** par défaut, la commande **make** va chercher un fichier nommé **Makefile**. Pour exécuter **make** sur un fichier avec un nom différent utiliser la commande : `make -f myMakefile`

**Exercice:** récupérez le projet qui se trouve dans le répertoire serie1\_0 du fichier archive dans le Topic1 (copiez ce répertoire chez vous)

## Essayez d'écrire le makefile sans utiliser des variables macros

Un Makefile possible pourrait être :

```
# Makefile

all: eatFruit

fruits.o : fruits.cc fruits.h
    g++ -c fruits.cc -o fruits.o

persons.o : persons.cc persons.h fruits.h
    g++ -c persons.cc -o persons.o

eatFruit.o : fruits.h persons.h eatFruit.cc
    g++ -c eatFruit.cc -o eatFruit.o

eatFruit : fruits.o persons.o eatFruit.o
    g++ eatFruit.o persons.o fruits.o -o eatFruit
```

Avec un tel **Makefile**, notre projet peut dès lors être compilé au moyen de la commande **make**. On remarque que la cible **all** n'est dans ce cas qu'un simple alias pour la cible **eatFruit** (c'est à dire que les deux noms sont équivalents).

Pour construire cette cible, **make** doit en premier lieu construire les cibles indiquées comme dépendances (aussi appelées les *pré-requis*); remarquons au passage que **make** ne (re)construit une cible que si *l'une au moins de ses dépendances est plus récente que la cible elle-même*.

C'est ce mécanisme qui permet à **make** de ne compiler que ce qui est strictement nécessaire. Ainsi, si l'on exécute la commande **make** une seconde fois, après la première compilation, le programme signalera :

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

De la même manière, si l'on venait à modifier le fichier **fruits.cc**, la commande **make** ne conduirait qu'à la recompilation de ce dernier (réalisation de la cible **fruits.o**, puisque l'un de ses pré-requis, **fruits.cc** correspond à un fichier dont la date de modification est postérieure à celle du fichier attaché à la cible), entraînant elle-même la mise à jour de la cible **eatFruit** (pour la même raison que précédemment).

Si l'on modifie par contre le fichier **fruits.h**, ce seront (toutes) les cibles **fruits.o**, **persons.o**, **eatFruit.o** et **eatFruit** qui seront mise à jour.

### Éléments avancés :

Supposons que nous souhaitions systématiquement préciser un certains nombres d'options au compilateur, pour permettre l'utilisation d'un dévermineur (-g), activer les warnings

(-Wall) et rendre le compilateur plus strict vis à vis du standard de C++ utilisé (-std=c++11). En plus, on veut remplacer dans chaque ligne la commande g++ avec une variable.

Plutôt que d'ajouter chacune de ces options à chaque commande de compilation (et devoir à nouveau tout remodifier lorsque l'on désirera supprimer les informations additionnelles ajoutées pour permettre le déverminage), il serait plus judicieux d'utiliser une variable (par exemple **CXXFLAGS**) pour mémoriser les options à transmettre au compilateur. Notre Makefile, devient alors :

```
# Makefile
CXX = g++
CXXFLAGS = -g -Wall -std=c++11

all: eatFruit

fruits.o : fruits.cc fruits.h
    $(CXX) $(CXXFLAGS) -c fruits.cc -o fruits.o

persons.o : persons.cc persons.h fruits.h
    $(CXX) $(CXXFLAGS) -c persons.cc -o persons.o

eatFruit.o : fruits.h persons.h eatFruit.cc
    $(CXX) $(CXXFLAGS) -c eatFruit.cc -o eatFruit.o

eatFruit : fruits.o persons.o eatFruit.o
    $(CXX) eatFruit.o persons.o fruits.o -o eatFruit
```

Ensuite, on veut utiliser une liste pour désigner les fichiers objets. Pour cela on utilise aussi une variable **OFILES**.

```
# Makefile
CXX = g++
CXXFLAGS = -g -Wall -std=c++11
OFILES = fruits.o persons.o eatFruit.o

all: eatFruit

fruits.o : fruits.cc fruits.h
    $(CXX) $(CXXFLAGS) -c fruits.cc -o fruits.o

persons.o : persons.cc persons.h fruits.h
    $(CXX) $(CXXFLAGS) -c persons.cc -o persons.o

eatFruit.o : fruits.h persons.h eatFruit.cc
    $(CXX) $(CXXFLAGS) -c eatFruit.cc -o eatFruit.o

eatFruit : $(OFILES)
    $(CXX) $(OFILES) -o eatFruit
```

La cible *particulière* **clean** est responsable de supprimer les fichiers inutiles (objets, exécutables, ...) qui s'accumulent dans le répertoire courant .

```
# Makefile
CXX = g++
CXXFLAGS = -g -Wall -std=c++11
OFILES = fruits.o persons.o eatFruit.o
```

```

all: eatFruit

fruits.o : fruits.cc fruits.h
          $(CXX) $(CXXFLAGS) -c fruits.cc -o fruits.o

persons.o : persons.cc persons.h fruits.h
           $(CXX) $(CXXFLAGS) -c persons.cc -o persons.o

eatFruit.o : fruits.h persons.h eatFruit.cc
            $(CXX) $(CXXFLAGS) -c eatFruit.cc -o eatFruit.o

eatFruit : $(OFILES)
           $(CXX) $(OFILES) -o eatFruit

clean:
        @echo " *** EFFACE MODULES OBJET ET EXECUTABLE ***"
        @/bin/rm -f *.o *.x *.cc~ *.h~ eatFruit

```

## Variables automatique

Ce qui a été présenté jusqu'à maintenant est suffisant pour vous permettre d'écrire un Makefile fonctionnel ; cependant, comme l'illustre l'exemple précédent, cette rédaction reste relativement fastidieuse. Les informations contenues dans cette section vont vous permettre d'augmenter considérablement le pouvoir expressif des instructions du **Makefile**, rendant ainsi sa rédaction plus aisée.

**make** maintient automatiquement à jour pour nous un certain nombre de **variables prédéfinies**, en les actualisant lors de l'exécution de chaque règle, en fonction de la cible concernée et de ses dépendances.

Parmi ces variables, citons:

<b>\$@</b>	<b>cible</b> de la règle courante.
<b>\$&lt;</b>	<b>première dépendance</b> (premier pré-requis de la liste des pré-requis)
<b>\$?</b>	<b>liste de toutes les dépendances</b> (séparées par une espace) <b>plus récentes que la cible courante</b> (les dépendances impliquant la mise à jour de la cible).
<b>\$^</b>	<b>liste de toutes les dépendances</b> (séparée par une espace) de la cible. Si un pré-requis est présent plusieurs fois dans une même liste de dépendance, il ne sera reporté qu'une fois par <b>\$^</b> .
<b>\$+</b>	<b>liste exacte de toutes les dépendances</b> (séparée par une espace) de la cible (à l'inverse de <b>\$^</b> , les pré-requis sont reportés par <b>\$+</b> autant de fois qu'ils sont indiqués dans la liste des dépendances).

Au vu de ce qui précède, on pourrait donc réécrire comme suit le Makefile :

```
# Makefile
```

```

CXX = g++
CXXFLAGS = -g -Wall -std=c++11
OFILES = fruits.o persons.o eatFruit.o

all: eatFruit

fruits.o : fruits.cc fruits.h
    $(CXX) $(CXXFLAGS) -c $< -o $@

persons.o : persons.cc persons.h fruits.h
    $(CXX) $(CXXFLAGS) -c $< -o $@

eatFruits.o : eatFruit.cc fruits.h persons.h
    $(CXX) $(CXXFLAGS) -c $< -o $@

eatFruit : $(OFILES)
    $(CXX) $(OFILES) -o $@

clean:
    @echo " *** EFFACE MODULES OBJET ET EXECUTABLE ***"
    @/bin/rm -f *.o *.x *.cc~ *.h~ eatFruit

```

Solution : graphe des dépendances:

