La programmation modulaire

Tristan Colombo 2002

Dans le cadre de votre projet de jeu de dames il va falloir répartir le code dans des fichiers « en-tête » (ayant l'extension .h). Il faut que les fonctions présentes au sein d'un même fichier aient un lien sémantique entre elles (toutes les primitives de base dans primitives.h, l'algorithme du min-max dans min_max.h, …).
Pour définir un fichier « en-tête » à partir d'un fichier .c, il suffit de copier toutes les déclarations de variables globales, de structures et les include dans un fichier .h puis de donner dans ce même fichier la déclaration de chaque fonction. Pour éviter une double inclusion de fichiers « en-tête » on peut utiliser l'astuce suivante (ici exemple sur un fichier dames.h) :

```
#ifndef DAMES_H
#define DAMES_H

   /* ici le contenu du fichier header */

#endif
```

Le fait de séparer notre programme en différents modules va nous permettre d'utiliser la *compilation séparée* grâce au programme make.
La structure d'un fichier Makefile est la suivante :
Cible: liste de dépendances
<tab> commandes UNIX
Par exemple, si l'on a le fichier Makefile suivant :

```
dames: dames.c dames.h interface.c
	gcc -o dames -O3 -Wall dames.c interface.c

clean:
	rm *.o dames

dames.o: dames.c dames.h
	gcc -c -O3 -Wall dames.c

interface.o: interface.c dames.c dames.h
	gcc -c -O3 -Wall interface.c
```

Avec ce fichier, deux options sont disponibles : on peut taper make ou make dames qui déclencheront la même routine de compilation, ou alors make clean qui déclenche la commande UNIX : `rm *.o dames`

Il est possible de définir des variables :

```
CC = gcc
FLAGS = -c -O3 -Wall

dames: dames.c dames.h interface.c
	$(CC) -o dames -O3 -Wall dames.c interface.c

...

dames.o: dames.c dames.h
	$(CC) $(FLAGS) dames.c
```

Il existe également des variables prédéfinies :

- $@ désigne le fichier cible courant
- $* désigne le fichier cible courant privé de son suffixe
- $< désigne le fichier qui a provoqué l'action

A partir de ces variables on peut définir des règles de compilation. Par exemple, pour définir les règles de compilation de tous les fichiers .o on peut faire :

```
.SUFFIXES: .o

# (Les remarques sont précédées du signe dièse)
# Règle de production d'un fichier .o
.c.o:; gcc –o $@ -c –O3 $<
```

Ainsi, s'il existe un fichier toto.c, la ligne précédente devient gcc –o toto.c –c –O3 toto.o