Introduction à Make
et aux fonctions pseudo-aléatoires

Support de TP C

1 Introduction à Make

Make est un outil de programmation, dont une version est disponible pour les systèmes Unix sous la licence GNU (c'est-à-dire libre à l'utilisation, la copie et la redistribution). Il permet d'automatiser des tâches, telles que la compilation de programmes ou l'archivage de fichiers. Pour fonctionner, Make a besoin d'un certain nombre d'informations qu'il cherchera par défaut dans un fichier nommé `makefile` ou `Makefile` (il est ainsi recommandé d'utiliser ces noms de fichiers). Le but de cette section est de donner les informations nécessaires à la création de fichiers `Makefile` pour la compilation de projets C peu complexes.

1.1 Syntaxe

Un fichier `Makefile` est constitué de plusieurs règles, chacune définie par la syntaxe suivante (une tabulation *(tab)* doit nécessairement précéder chaque commande) :

```
cible1 : [dépendance1] [dépendance2] ...
(tab) [commande1]
(tab) [commande2]
...

cible2 : [dépendanceA] [dépendanceB] ...
(tab) [commandeA]
(tab) [commandeB]
...
```

La commande shell `make cible1` entraîne l'évaluation successive de chacune de ses dépendances (`dépendance1`, `dépendance2`, etc...). Pour une compilation par exemple, Make compare la date de modification de la cible avec celle de chaque dépendance : si au moins une dépendance s'avère plus récente que la cible, les commandes de la cible (`commande1`, `commande2`, etc...) sont exécutées.

Si une dépendance est la cible d'une autre règle du `Makefile`, cette cible est à son tout évaluée, et, si nécessaire, ses commandes sont exécutées avant celles de la cible originale. Par exemple, si `dépendance2` est égale à `cible2` (et si une dépendance de `cible2` est plus récente que `cible2`), `make cible1` entraînera l'exécution successive des commandes `commandeA`, `commandeB`, etc... puis `commande1`, `commande2`, etc...

Si une cible ne contient pas de dépendance, ses commandes sont exécutées à chaque appel de la cible.

Pour éviter d'écrire de longues commandes ou une longue liste de dépendance sur la même ligne, on peut utiliser le caractère de continuation de ligne \.

1.2 Exemple simple

On considère un programme `etudiant_INSA.c` qui utilise les couples de fichiers (`.c` et `.h`) : `traîner_sur_internet` et `manger_des_pâtes`. Le fichier `Makefile` suivant permet de compiler l'ensemble des fichiers par la commande shell `make etudiant_INSA`.

```
etudiant_INSA : etudiant_INSA.o \
            traîner_sur_internet.o manger_des_pâtes.o
(tab) gcc etudiant_INSA.o traîner_sur_internet.o \
            manger_des_pâtes.o -o etudiant_INSA

etudiant_INSA.o : etudiant_INSA.c
(tab) gcc -c etudiant_INSA.c

traîner_sur_internet.o : traîner_sur_internet.c
(tab) gcc -c traîner_sur_internet.c

manger_des_pâtes.o : manger_des_pâtes.c
(tab) gcc -c manger_des_pâtes.c
```

1.3 Cibles conventionnelles

Les cibles `default`, `all`, `clean` et `install` se retrouvent couramment dans un `Makefile`. C'est souvent une bonne idée de les implanter, lorsqu'elles sont applicables (`make install`, par exemple, n'est généralement pas nécessaire dans un TP).

- `default` : décrit les cibles évaluées et exécutées par la commande shell `make` appelée sans argument,
- `all` : réalise l'ensemble des cibles décrites par le fichier `Makefile` (par exemple, compile l'ensemble des fichiers d'un projet C),
- `clean` : supprime les fichiers créés par l'exécution de Make, par exemple pour repartir d'un environnement "propre" (en C, `make clean` supprime généralement les fichiers objets et les exécutables)

– `install` : installe une application ou un programme, idéalement au bon endroit (le problème vient souvent de ce que le "bon endroit" dépend du système qu'on utilise).

1.4 Règles génériques

Une règle générique ne mentionne pas de cibles particulières, mais porte sur un ensemble de cibles. En C, une règle générique permet généralement de décrire la compilation de fichiers sources `.c` en fichiers objets `.o`. Les variables automatiques `$^` et `$<` sont souvent utilisées pour cela : `$^` représente l'ensemble des dépendances de la règle, et `$<` représente uniquement la première dépendance. `$@` est une autre variable automatique souvent utilisée, représentant le nom de la cible dans une règle.

Ci-dessous, un exemple de règle générique, créant les fichiers objets à partir des fichiers sources de même nom. La variable automatique `%` y est utilisé pour représenter la même valeur du côté de la cible et du côté des dépendances.

```
%.o : %.c :
(tab) gcc -c $<
```

1.5 Exemple plus concret de `Makefile`

En reprenant le programme `etudiant_INSA` et les fichiers dont il dépend `traîner_sur_internet` et `manger_des_pâtes`, on obtient :

```
default : etudiant_INSA

all : etudiant_INSA

clean :
(tab) rm *.o etudiant_INSA

etudiant_INSA : etudiant_INSA.o \
             traîner_sur_internet.o manger_des_pâtes.o
(tab) gcc etudiant_INSA.o traîner_sur_internet.o \
             manger_des_pâtes.o -o $@

%.o : %.c
(tab) gcc -c $<

traîner_sur_internet.o : traîner_sur_internet.h

manger_des_pâtes.o : manger_des_pâtes.h
```

2 Fonctions pseudo-aléatoires

Les ordinateurs et les programmes informatiques ont un comportement déterministes, et sont ainsi incapables de générer par eux-mêmes des nombres choisis aléatoirement. Pour résoudre partiellement ce problème, on utilise des générateurs pseudo-aléatoires, qui construisent une série de nombres pseudo-aléatoires à partir d'une graine donnée. La même graine, utilisée plusieurs fois, donnera à chaque fois la même série de nombres pseudo-aléatoires. De plus, à partir d'une certaine quantité de nombres générés, la série de nombres devient prédicitble, et ne peut plus être considérée comme aléatoire. Malgré ces limites, la génération de nombres pseudo-aléatoires reste un moyen simple et efficace pour introduire du (pseudo) hasard dans les systèmes informatiques. Cette section présente rapidement deux fonctions de la librairie standard C (`stdlib`) permettant de générer des nombres pseudo-aléatoires : `rand` et `srand`.

2.1 Les fonctions `rand` et `srand`

La fonction `void srand(unsigned int seed)` a pour but d'initialiser une série de nombres pseudo-aléatoires à partir de la graine `seed` passée en argument. La série pseudo-aléatoire utilisée par défaut, c'est-à-dire quand la fonction `srand` n'a pas été utilisée pour fournir une graine, a une graine de 1.

Une fois la série pseudo-aléatoire initialisée par une graine, chaque appel de la fonction `int rand()` renvoie la valeur suivante dans la série. Cette valeur est comprise entre 0 et la valeur RAND_MAX définie dans `stdlib`.

Pour avoir un nombre aléatoire compris entre 0 et 1, donc facilement assimilable a une probabilité, il suffit de diviser par RAND_MAX le nombre aléatoire retourné par `rand()`. Ces nombres étant de type `integer`, il est nécessaire de les forcer au type `float` pour ne pas se retrouver avec le résultat d'une division entière.

2.2 Exemple d'utilisation

Le programme suivant affiche les 10 premiers nombres de la série pseudo-aléatoire générée avec une graine de valeur 4, ainsi que les valeurs associées normées entre 0 et 1. Définir la valeur de la graine dans le programme, comme ici, n'est généralement pas la démarche à suivre, puisque chaque exécution du programme donnera la même succession de nombres.

```
# include <stdlib.h>
# include <stdio.h>

int main () {
```

```
  int loop, random_number;
  int seed = 4;
  float normed;

  srand(seed);
  for (loop = 1; loop <=10; loop++) {
    random_number = rand();
    printf("%d : ", random_number);
    normed = (float) random_number / (float) RAND_MAX;
    printf("%f\n", normed);
  }
  return 0;
}
```

2.3 Comment choisir une graine ?

C'est la question prinicpale dans l'utilisation de `rand` et `srand`. Différentes solutions sont possibles. Sous Unix, on choisit généralement soit de prendre les microsecondes du temps courant (en utilisant la fonction `gettimeofday`), soit de prendre le pid du processus exécutant le programme (en utilisant la fonction `getpid`).

Dans l'exemple suivant, on choisit d'utiliser la fonction `int gettimeofday (struct timeval *tv, struct timezone *tz)`. Cette fonction prend en entrée deux structures par adresse : une `struct timeval` contenant les champs `tv_sec` et `tv_usec` (respectivement les secondes et les microsecondes du temps mis à jour par le système d'exploitation), et une `struct timezone`, généralement obsolète, dont ici nous ne nous soucions pas.

```
# include <stdlib.h>
# include <stdio.h>
# include <sys/time.h>

int main () {
  int loop, random_number;
  float normed;
  struct timeval tv;

  gettimeofday(&tv, NULL);
  srand(tv.tv_usec);
  for (loop = 1; loop <=10; loop++) {
    random_number = rand();
    printf("%d : ", random_number);
    normed = (float) random_number / (float) RAND_MAX;
    printf("%f\n", normed);
  }
  return 0;
}
```