Informatique pour l'entreprise

Make

Marie Pelleau & Olivier Baldellon
marie.pelleau@univ-cotedazur.fr,
olivier.baldellon@univ-cotedazur.fr

13 février 2023

Outil make

- Vocation de `make` = gérer la construction de logiciels modulaires
- Réaliser des compilations (ou toute autre action) dans un certain ordre (compatible avec des règles de dépendance) : fichier **makefile**
- Dans le cas où `make` réalise des actions autres que la compilation, cet outil est équivalent à un script SHELL, si ce n'est qu'il y a la gestion des règles de dépendance en plus
- `make` ne produit les cibles que si les cibles dont elles dépendent sont plus récentes qu'elles

Outil make

- Les IDE créent des makefile
- Visual Studio
  - Positionnez la souris sur un projet, puis faites un clic droit, puis Propriétés
  - Dans la partie gauche sous C/C++ et Linker il y a une entrée Command Line : elle permet de voir la commande qui est effectivement appelée pour compiler et pour linker

Utilitaire make et makefile

- Existe partout (`make` sur Linux, `nmake` sur windows)
- Exécute une suite d'instructions contenues dans un fichier dit "makefile"
- Souvent le fichier "makefile" s'appelle `Makefile`
- Structure du fichier

```
entree : dépendances
    action à réaliser
```

- Attention tabulations **importantes** avant actions

Structure générale du makefile

- Ce fichier de texte contient une suite d'entrées qui spécifient les dépendances entre les fichiers
- Il est constitué de lignes logiques (une ligne logique pouvant être une ligne physique ou plusieurs lignes physiques séparées par le signe \)
- Les commentaires débutent par le signe # et se terminent à la fin de ligne
- Contenu (ordre recommandé) :
  - définitions de macros
  - règles implicites
  - règles explicites ou entrées

Règles explicites

$\texttt{cible}_1$ ... $\texttt{cible}_m$: $\texttt{dépendance}_1$ ... $\texttt{dépendance}_n$
$\texttt{action}_1$
$\texttt{action}_2$
...
$\texttt{action}_p$

- Traduction : mettre à jour les cibles : $\texttt{cible}_1$ ... $\texttt{cible}_m$ quand les dépendances $\texttt{dépendance}_1$ ... $\texttt{dépendance}_n$ sont modifiées en effectuant les opérations $\texttt{action}_1$, $\texttt{action}_2$ ... $\texttt{action}_p$
- La première entrée est la cible principale

Exemple

math.h

```
/* Retourne a^b (ou 1.0 si b < 0) */
double puissance (int , int);
```

math.c

```
#include "math.h"

double puissance (int a, int b) {
    double z = 1.0;
    while (b > 0) {
        z *= a;
        b--;
    }
    return z;
}
```

Exemple

essai.c

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

int main (int argc, char *argv[]) {
    int x, y;
    if (argc != 3) {
        fprintf(stderr, "usage: %s x y >= 0 (x^y)\n", argv[0]);
        return 1;
    }
    x = atoi(argv[1]);
    y = atoi(argv[2]);
    if (y < 0) {
        fprintf(stderr, "usage: %s x y >= 0 (x^y)\n", argv[0]);
        return 2;
    }
    printf("x = %d, y = %d, x^y = %.2f\n", x, y, puissance(x, y));
    return 0;
}
```

Exemple

Remarque : Si `math.o` n'est pas "lu" lors de l'édition de liens, il y aura une référence non résolue de la fonction puissance

```
% gcc essai.c
...: In function "main":
...: undefined reference to "puissance":
...: ld returned 1 exit status
```

Exemple

Makefile

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJECTS = math.o essai.o

vasy : $(OBJECTS)
        $(CC) -o vasy $(OBJECTS)
        echo "La compilation est finie, l'exécutable vasy est créé"

essai.o : essai.c math.h
        echo "Compilation de essai.c"
        $(CC) -c $(CFLAGS) essai.c

math.o : math.c math.h
        echo "Compilation de math.c"
        $(CC) -c $(CFLAGS) math.c

clean :
        rm -f $(OBJECTS) vasy *~

print :
        lpr math.h math.c essai.c
```

makefile

- Si une action s'exécute sans erreur (code de retour nul), make passe à l'action suivante de l'entrée en cours, ou à une autre entrée si l'entrée en cours est à jour
- Si erreur (et - absent), make arrête toute exécution

- Les actions peuvent être précédées des signes suivants :
  - `-` : si l'action s'exécute avec un code de retour anormal (donc erreur), make continue
  - `@` : l'impression de la commande elle-même est supprimée
  - `@-`, `-@` : pour combiner les précédents

makefile : exemple

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJECTS = math.o essai.o

vasy : $(OBJECTS)
        @$(CC) -o vasy $(OBJECTS)
        @echo "La compilation est finie, l'exécutable vasy est créé"

essai.o : essai.c math.h
        @echo "Compilation de essai.c"
        @$(CC) -c $(CFLAGS) essai.c

math.o : math.c math.h
        @echo "Compilation de math.c"
        @$(CC) -c $(CFLAGS) math.c

clean :
        -rm -f $(OBJECTS) vasy *~

print :
        lpr math.h math.c essai.c
```

Commandes usuelles

Il est bien pratique d'avoir des entrées d'impression, de nettoyage ou d'installation

```
impression :
-lpr *.c *.h

menage :
@-rm *.o *.out core *~

install :
mv a.out /usr/bin/copy
chmod a+x /usr/bin/copy
```

Appel de make

`make [-f nom_du_makefile] [options] [nom_des_cibles]`

Options :

- `-f` : si option manquante, `make` prendra comme fichier de commandes un des fichiers `makefile`, `Makefile`, `s.makefile` ou `s.Makefile` (s'il le trouve dans le répertoire courant)
- `-d` : permet le mode "Debug", c'est-à-dire écrit les informations détaillées sur les fichiers examinés ainsi que leur date
- `-n` : imprime les commandes qui auraient dû être exécutées pour mettre à jour la cible principale (mais ne les exécute pas)
- `-t` : permet de mettre à jour les fichiers cible

Appel de make

`make [-f nom_du_makefile] [options] [nom_des_cibles]`

Options :

- `-p` : affiche l'ensemble complet des macros connues par `make`, ainsi que la liste des suffixes et de leurs règles correspondantes
- `-s` : n'imprime pas les commandes qui s'exécutent; `make` fait son travail en silence
- `-S` : abandonne le travail sur l'entrée courante en cas d'échec d'une des commandes relatives à cette entrée (L'option opposée est `-k`)
- `nom_des_cibles` : si aucun nom n'est donné, la cible principale sera la première entrée explicite du makefile

Appel de make

Exemple

```
make
make math.o
make clean
make vasy
```

Macros

Définition de macros

- Syntaxe
  - `chaîne1 = chaîne2`
  - chaîne2 est une suite de caractères se terminant au caractère # de début de commentaire ou au caractère de fin de ligne (s'il n'est pas précédé du caractère d'échappement \)
- Dans la suite du makefile, chaque apparition de `$(chaîne1)` sera remplacée par `chaîne2`
- Exemples
  - `OBJETS = f1.o f2.o f3.o`
  - `SOURCES = f1.h f1.c f2.h f2.c f3.h f3.c`
  - `REPINST = /usr/bin`

Macros

Remplacement d'une sous-chaîne par une autre dans une chaîne

- Syntaxe
  - `$(chaîne:subst1=subst2)`
  - `subst1` est remplacé par `subst2` dans chaîne
- Exemples
  - `$(OBJETS:f2.o=)`
  - `$(OBJETS:f2.o=fn.o)`
  - `$(REPINST:bin=local/bin)`

Macros internes

- `$*` : le nom de la cible courante sans suffixe
- `$@` : le nom complet de la cible courante
- `$<` : la première dépendance
- `$^` : la liste complète des dépendances
- `$?` : la liste des dépendances plus récentes que la cible

makefile : exemple

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJECTS = math.o essai.o

vasy : $(OBJECTS)
        @$(CC) -o vasy $^
        @echo "La compilation est finie, l'exécutable $@ est créé"

essai.o : essai.c math.h
        @echo "Compilation de $*.c"
        @$(CC) -c $(CFLAGS) $*.c

math.o : math.c math.h
        @echo "Compilation de $*.c"
        @$(CC) -c $(CFLAGS) $*.c

clean :
        -rm -f $(OBJECTS) vasy *~

print :
        lpr math.h math.c essai.c
```

- Les variables d'environnement sont supposées être des définitions de macros
  - Les variables d'environnement l'emportent sur les macros internes définies par défaut
  - Les macros définies dans le makefile l'emportent sur les variables d'environnement
  - Les macros définies dans une ligne de commande l'emportent sur les macros définies dans le makefile
- L'option -e change tout ça de telle façon que les variables d'environnement l'emportent sur les macros définies dans le makefile

Règles implicites

- Elles servent à donner les actions communes aux fichiers se terminant par le même suffixe
  `.SUFFIXES : liste de suffixes`
  `.source.cible :`
  `actions`
- Dans `.SUFFIXES:` on définit les suffixes standard utilisés par les outils pour identifier des types de fichiers particuliers
- Traduction : À partir de `XX.source`, on produit `XX.cible` grâce à `actions`
- Pour supprimer les règles implicites par défaut, appeler make avec l'option `-r`, ou écrire `.SUFFIXES:` seulement

Règles implicites

Exemple

Pour tous les fichiers sources C (ayant comme suffixe `.c`), on appelle le compilateur C avec l'option `-c`

```
.SUFFIXES : .out .o .h .c
.c.o :
gcc -c -Wall -pedantic -ansi $*.c
```

Exemple : règles implicites

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJECTS = math.o essai.o

vasy : $(OBJECTS)
        @$(CC) -o vasy $^
        @echo "La compilation est finie, l'exécutable $@ est créé"

.c.o :
        @echo "Compilation de $*.c"
        @$(CC) -c $(CFLAGS) $*.c

essai.o : essai.c math.h

math.o : math.c math.h

clean :
        -rm -f $(OBJECTS) vasy *~

print :
        lpr math.h math.c essai.c
```

Exemple : règles implicites par défaut

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJECTS = math.o essai.o

vasy : $(OBJECTS)
        @$(CC) -o vasy $^
        @echo "La compilation est finie, l'exécutable $@ est créé"

essai.o : essai.c math.h

math.o : math.c math.h

clean :
        -rm -f $(OBJECTS) vasy *~

print :
        lpr math.h math.c essai.c
```

Exemple : avec demande à l'utilisateur

```
CC = gcc
CFLAGS = -c -Wall -pedantic -ansi
OBJECTS = math.o essai.o

vasy : $(OBJECTS)
        @$(CC) -o vasy $^
        @echo "La compilation est finie, l'exécutable $@ est créé"

.c.o :
        @echo "avec Debug ?"
        @-read REP; \
        case $${REP} in \
        y|Y|o|O) $(CC) $(CFLAGS) -g -o $@ $*.c;; \
        n|N) $(CC) $(CFLAGS) -o $@ $*.c;; \
        *) echo "$*.c non compilé";; \
        esac

essai.o : essai.c math.h

math.o : math.c math.h

clean :
        -rm -f $(OBJECTS) vasy *~

print :
        lpr math.h math.c essai.c
```