Programmation

Génération de documentation

Erwan Guillou

UFR Informatique
Université Claude Bernard Lyon1

15 octobre 2008

Plan du cours

1. Bases
   - Définitions
   - L'utilitaire make
   - Exemple de makefile
2. Les macros d'un makefile
   - Macros
   - Sur l'exemple
3. Utilisation avancée
   - Règles d'inférence
   - Cibles sans dépendances
   - Réécriture de macros
   - Génération automatique des dépendances

Définitions

- Cible : une action à accomplir.
- Dépendances (de la cible) : fichiers nécessaires pour générer la cible
- Actions (sur les dépendances) actions à appliquer aux dépendance pour générer la cible
- Exemples
  - une cellule de tableur et les cellules auxquelles elle fait référence.
  - fichier postscript et fichier(s) permettant de le créer.
  - application et les fichiers exécutables, fichiers exécutables et fichiers sources (.c et .h).

Exemple

```
cible : dep1 dep2 ... depn
        actions pour produire la cible
dep1 : dep1.1 dep1.2 ... dep1.m1
        actions pour produire dep1
dep2 : dep2.1 dep2.2 ... dep2.m1
        actions pour produire dep2
.
.
.
depn : depn.1 depn.2 ... depn.m1
        actions pour produire depn
```

Les bases

- Le fichier de fabrication (*makefile*, *Makefile* ou autre) traduit le graphe de dépendances.
- L'utilitaire **make** exécute ces actions dans un ordre respectant les dépendances.
- Ne sont actualisées que les cibles qui sont plus anciennes que leurs dépendances.
- **make** exécute la première des cible du fichier de commande, en l'absence de toute autre indication.
- **make nom-de-tâche** exécute la cible spécifiée, si elle est bien présente dans le fichier de fabrication.

Paramètres de make

- **make -f fichier** exécute make sur le fichier de dépendances fichier.
- **make -n** tâche permet de lister les actions nécessaires à la réalisation d'une tâche cible, mais sans les effectuer.
- **make -k** abandonne la cible courante en cas d'erreur, et passe à la suivante.
- **make -s** mode silencieux ; pas d'affichage des tâches avant exécution.
- **make -p** liste la totalité des macros-définitions et des cibles.
- **make -C dir target** permet la descente dans un répertoire donné et de lancer make sur la cible demandée.

Les fichiers du projet

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void Hello ( void )
5  {
6          printf ( "Hello World !\n" );
7  }
```

```
1  #ifndef __HELLO__
2  #define __HELLO__
3
4  void Hello ( void );
5
6  #endif
```

```
1  #include "Hello.h"
2
3  int main ( void )
4  {
5          Hello ();
6  }
```

Makefile minimal

```
1 hello: hello.o main.o
2 	gcc -o hello hello.o main.o
3
4 hello.o: hello.c hello.h
5 	gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic
6
7 main.o: main.c hello.h
8 	gcc -o main.o -c main.c -W -Wall -ansi -pedantic
```

Makefile avancé

```
1  all: hello
2
3  hello: hello.o main.o
4         gcc -o hello hello.o main.o
5
6  hello.o: hello.c hello.h
7         gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic
8
9  main.o: main.c hello.h
10        gcc -o main.o -c main.c -W -Wall -ansi -pedantic
11
12 clean:
13        rm -rf *.o
14
15 distclean: clean
16        rm -rf hello
```

Macros définies par l'utilisateur

- Définition : **NOM = valeur** (où les deux sont des chaînes de caractères).
- Exemple : *SOURCES = f1.c f2.c*
- Utilisation : **$(NOM)** ou éventuellement **$N** si le nom est réduit à un caractère.
- Exemple : *gcc -c $(SOURCES)*

Macros définies automatiquement

- **$@** désigne la cible courante.
- **$∧** chaîne de caractères composée des noms de toutes les dépendances.
- **$?** chaîne de caractères composée des noms des dépendances plus récentes que la cible.
- **$<** contient le nom de la dépendance permettant la génération de la cible (la première dans la liste).
- **$*** le préfixe du nom de fichier partagé entre la cible courante et la dépendance.

Makefile avec macros

```
1  BIN = hello
2  OBJS = hello.o main.o
3  CFLAGS = -W -Wall -ansi -pedantic
4
5  all: $(BIN)
6
7  $(BIN): $(OBJS)
8          gcc -o $@ $(OBJS)
9
10 hello.o: hello.c hello.h
11         gcc -o $@ -c $< $(CFLAGS)
12
13 main.o: main.c hello.h
14         gcc -o $@ -c $< $(CFLAGS)
15
16 clean:
17         rm -rf $(OBJS)
18
19 distclean: clean
20         rm -rf $(BIN)
```

Règles d'inférence

- Dans l'exemple précédent, plusieurs règles réalisent la même action.
- Il est possible d'écrire des règles génériques (par exemple construire un *.o* à partir d'un *.c*) qui se verront appelées par défaut.
- l'écriture de telles règles se fait comme suit :

```
1 %.o: %.c
2         commandes
```

Makefile avec règles d'inférence

```
1  BIN = hello
2  OBJS = hello.o main.o
3  CFLAGS = -W -Wall -ansi -pedantic
4
5  all: $(BIN)
6
7  $(BIN): $(OBJS)
8          gcc -o $@ $(OBJS)
9
10 %.o: %.c
11         gcc -o $@ -c $< $(CFLAGS)
12
13 clean:
14         rm -rf $(OBJS)
15
16 distclean: clean
17         rm -rf $(BIN)
```

- Problème : on perd la dépendance sur les fichiers de header
- Solution : ajouter les dépendances

Makefile avec règles d'inférence et dépendances

```
1  CC=gcc
2  LD=gcc
3  BIN = hello
4  OBJS = hello.o main.o
5  CFLAGS = -W -Wall -ansi -pedantic
6
7  all: $(BIN)
8
9  $(BIN): $(OBJS)
10         $(LD) -o $@ $(OBJS)
11
12 %.o: %.c
13         $(CC) -o $@ -c $< $(CFLAGS)
14
15 main.o: hello.h
16 hello.o: hello.h
17
18 clean:
19         rm -rf $(OBJS)
20
21 distclean: clean
22         rm -rf $(BIN)
```

Cibles sans dépendances

- Dans l'exemple précédent, **clean** est une cible ne présentant pas de dépendance
- Dans le projet, il existe un répertoire nommé **clean**
- Celui-ci sera forcément plus récent que ses dépendances
- la règle ne sera jamais exécutée
- Solution : il existe une cible particulière **.PHONY** dont les dépendances seront toujours reconstruites

Makefile avec règles .PHONY

```
1  CC=gcc
2  LD=gcc
3  BIN = hello
4  OBJS = hello.o main.o
5  CFLAGS = -W -Wall -ansi -pedantic
6
7  all: $(BIN)
8
9  $(BIN): $(OBJS)
10         $(LD) -o $@ $(OBJS)
11
12 %.o: %.c
13         $(CC) -o $@ -c $< $(CFLAGS)
14
15 main.o: hello.h
16 hello.o: hello.h
17
18 .PHONY: clean distclean
19
20 clean:
21         rm -rf $(OBJS)
22
23 distclean: clean
24         rm -rf $(BIN)
```

Réécriture de macros

- Plutôt que d'énumérer la liste des fichiers objets
- Il est possible de la générer automatiquement à partir de la liste des fichiers source
- On utilise une réécriture de variable :

```
1 $(VARIABLE:A_MODIFIER=TEXTE_DE_REMPLACEMENT)
```

- De plus, la liste des fichiers source peut être générée automatiquement :

```
1 $(wildcard motif de recherche)
```

Makefile avec réécriture de variable

```
1  CC=gcc
2  LD=gcc
3  BIN = hello
4  SRCS=$(wildcard *.c)
5  OBJS = $(SRCS:.c=.o)
6  CFLAGS = -W -Wall -ansi -pedantic
7
8  all: $(BIN)
9
10 $(BIN): $(OBJS)
11         $(LD) -o $@ $(OBJS)
12
13 %.o: %.c
14         $(CC) -o $@ -c $< $(CFLAGS)
15
16 main.o: hello.h
17 hello.o: hello.h
18
19 .PHONY: clean distclean
20
21 clean:
22         rm -rf $(OBJS)
23
24 distclean: clean
25         rm -rf $(BIN)
```

Génération des dépendances

- Plus le projet grossi, plus il y a de fichiers
- les dépendances sont donc lourdes à écrire
- Il est possible de les générer automatiquement à l'aide du compilateur :

```
1 DEPS=$(SRCS:%.c=%.d)
2 %.d: %.c
3         $(CC) $(CFLAGS) -MM -MD -o $@ $<
```

- Il n'y a plus qu'à les inclure :

```
1 -include $(DEPS)
```

Makefile final

```
1  CC=gcc
2  LD=gcc
3  BIN = hello
4  SRCS=$(wildcard *.c)
5  OBJS = $(SRCS:.c=.o)
6  DEPS = $(SRCS:.c=.d)
7  CFLAGS = -W -Wall -ansi -pedantic
8
9  all: $(BIN)
10
11 $(BIN): $(OBJS)
12         $(LD) -o $@ $(OBJS)
13
14 %.o: %.c
15         $(CC) -o $@ -c $< $(CFLAGS)
16 %.d: %.c
17         $(CC) $(CFLAGS) -MM -MD -o $@ $<
18
19 -include $(DEPS)
20
21 .PHONY: clean distclean
22
23 clean:
24         rm -rf $(OBJS)
25 distclean: clean
26         rm -rf $(BIN)
```