Tutoriel Basique sur les Makefile

Introduction

Un **Makefile** est un fichier texte utilisé par l'outil make pour automatiser la compilation et la gestion des dépendances dans les projets logiciels, principalement en C/C++. Il permet de simplifier le processus de construction en définissant des règles pour compiler et lier les fichiers.

Ce tutoriel vous guidera à travers les bases pour écrire et utiliser un Makefile.

Structure d'un Makefile

Un Makefile est constitué de plusieurs parties :

1. **Cibles (Targets)** : Ce que vous voulez produire (par exemple, un exécutable).
2. **Prérequis (Dependencies)** : Les fichiers nécessaires pour produire la cible.
3. **Commandes (Commands)** : Les actions à exécuter pour produire la cible.

Syntaxe de Base

cible : prerequis
\tcommande

- Chaque commande doit commencer par une **tabulation** (pas des espaces).
- Une ligne vide ou sans tabulation sera interprétée comme une erreur.

Exemple Simple

```
all: programme

programme: main.o util.o
	gcc -o programme main.o util.o

main.o: main.c
	gcc -c main.c

util.o: util.c
	gcc -c util.c

clean:
	rm -f *.o programme
```

Explication

- **Cible all** : Une convention pour regrouper toutes les tâches principales.
- **Cible programme** : Dépend des fichiers objets main.o et util.o. La commande gcc les lie pour créer l'exécutable.
- **Cible main.o et util.o** : Dépendent de leurs fichiers source respectifs. Chaque fichier .c est compilé en fichier .o (fichier objet).
- **Cible clean** : Supprime les fichiers intermédiaires et l'exécutable.

Variables dans Makefile

Les variables permettent de simplifier et de réutiliser du code.

Définition et Utilisation

```
CC = gcc
CFLAGS = -Wall -Wextra

all: programme
```

```
programme: main.o util.o
	$(CC) $(CFLAGS) -o programme main.o util.o

main.o: main.c
	$(CC) $(CFLAGS) -c main.c

util.o: util.c
	$(CC) $(CFLAGS) -c util.c

clean:
	rm -f *.o programme
```

Explication

- CC : Définit le compilateur à utiliser (par défaut, gcc).
- CFLAGS : Options de compilation, comme -Wall pour activer les avertissements.
- Les variables sont référencées avec $(NOM_VARIABLE).

Règles Implicites

make utilise des règles implicites pour simplifier le Makefile. Par exemple, make sait automatiquement comment créer un fichier .o à partir d'un fichier .c.

Exemple Minimal

```
all: programme

programme: main.o util.o
	gcc -o programme main.o util.o

clean:
	rm -f *.o programme
```

Dans cet exemple, make déduit comment créer main.o et util.o à partir de main.c et util.c.

Gestion des Dépendances

Si un fichier source est modifié, make reconstruit uniquement les cibles affectées. Cependant, il est souvent nécessaire de gérer explicitement les dépendances pour les fichiers d'en-tête.

Exemple avec Fichiers d'En-tête

```
main.o: main.c util.h
util.o: util.c util.h
```

Ici, si util.h est modifié, make recompilera main.o et util.o.

Cibles Spéciales

.PHONY

Une cible **.PHONY** n'est pas associée à un fichier. Elle force l'exécution de la commande, même si un fichier portant le même nom existe.

Exemple

```
.PHONY: all clean

all: programme

clean:
	rm -f *.o programme
```

Utilisation Avancée

Compilation Parallèle

Avec l'option -j, vous pouvez exécuter plusieurs tâches en parallèle :

make -j4

Cela permet de réduire le temps de compilation.

Inclusion de Fichiers

Pour inclure des Makefiles secondaires :

include fichier.mk

Commandes Utiles

- **make** : Exécute les cibles définies.
- **make clean** : Exécute la cible clean.
- **make -n** : Affiche les commandes sans les exécuter (mode "dry-run").
- **make -j** : Active la compilation parallèle.

Exemple Complet

Structure de Projet

project/
|-- Makefile
|-- main.c
|-- util.c
|-- util.h

Contenu du Makefile

```
CC = gcc
CFLAGS = -Wall -Wextra

all: programme

programme: main.o util.o
	$(CC) $(CFLAGS) -o programme main.o util.o

main.o: main.c util.h
	$(CC) $(CFLAGS) -c main.c

util.o: util.c util.h
	$(CC) $(CFLAGS) -c util.c

clean:
	rm -f *.o programme

.PHONY: all clean
```

Conclusion

Un Makefile permet d'automatiser et d'optimiser la compilation des projets. Avec des variables, des règles implicites et des cibles personnalisées, il est possible de créer des workflows adaptés aux besoins des développeurs. Expérimentez pour maîtriser cet outil puissant !