POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

COMPILATION AVEC *MAKEFILES*

28 AOÛT 2023

CHARLES DE LAFONTAINE, JÉRÔME COLLIN, MERIAM BEN RABIA
GIGL | Polytechnique Montréal

N.B. Le masculin est utilisé pour alléger le texte.

TABLE DES MATIÈRES

COMPILATION AVEC *MAKEFILES*

1. DÉFINITION

Un *Makefile* est un fichier utilisé principalement avec l'outil « *make* » pour automatiser les tâches de compilation et de construction d'un programme. C'est une manière efficace de regrouper du code pour faciliter sa réutilisation. Les *Makefiles* vous évitent de taper de longues séries de commandes et vous permettent d'exécuter des tâches complexes avec une simple commande. Avant de programmer son premier *Makefile*, il est nécessaire de bien comprendre les concepts de librairies et de compilateur.

2. LIBRAIRIES OU BIBLIOTHÈQUES LOGICIELLES

Les librairies permettent de regrouper des morceaux de code que vous pouvez utiliser dans différents projets. Il existe deux types principaux :

2.1. Librairies statiques

Ces librairies sont intégrées directement dans l'exécutable final lors de la compilation. Elles sont souvent utilisées dans des microcontrôleurs où un seul exécutable est chargé.

Exemple

Imaginez avoir une librairie qui gère des opérations mathématiques basiques appelée **`math_ops`**. Vous pouvez la compiler en une librairie statique **`math_ops.a`** et la lier à différents programmes qui nécessitent ces opérations.

2.2. Librairies dynamiques

Ces librairies sont chargées en mémoire au moment de l'exécution. De nombreux exécutables peuvent partager la même librairie dynamique, réduisant ainsi l'utilisation de la mémoire.

Exemple

Si vous avez une librairie qui gère des interactions avec une base de données et est utilisée par plusieurs applications sur votre système, il serait plus efficient de l'avoir comme une librairie dynamique, par exemple **`bd_operations.so`**.

3. LE RÔLE DU COMPILATEUR

Il est important de comprendre que vous pouvez avoir différents types de compilateurs sur une machine. Par exemple, le cours *INF1900* utilise *GCC* comme base, mais pour certains microcontrôleurs, un compilateur croisé tel que *avr-gcc* est utilisé. Un compilateur croisé génère du code pour une architecture différente de celle du système sur lequel il tourne.

Exemple

Lorsque vous développez pour un microcontrôleur *Atmel AVR*, même si vous codez sur un système *Linux*, vous utiliserez *avr-gcc* pour compiler votre code pour le microcontrôleur plutôt que *GCC* qui compilerait pour votre machine *Linux*.

4. FONDAMENTAUX DES *MAKEFILES*

4.1. Organisation

Un *Makefile* par répertoire. Les règles communes peuvent être regroupées dans un fichier séparé et incluses dans d'autres *Makefiles*.

4.2. Compilation

Chaque *Makefile* ne compile que les sources présentes dans son répertoire et produit soit une librairie, soit un exécutable.

4.3. Nettoyage

Une règle « *clean* » est utilisée pour supprimer les fichiers générés.

4.4. Défaut

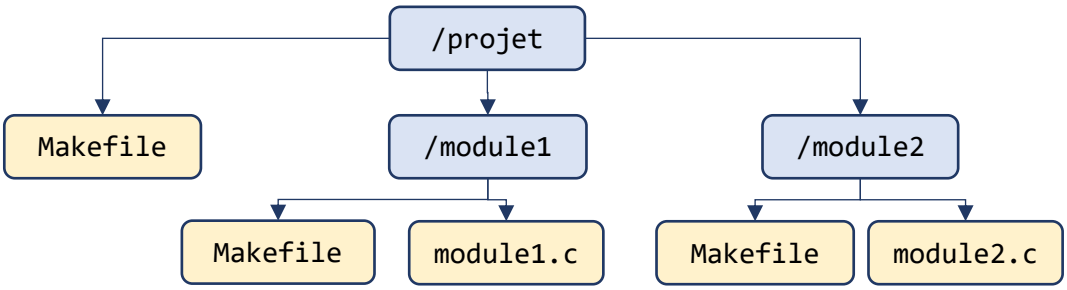
Une règle « *all* » est souvent utilisée comme règle par défaut.

4.5. Documentation et ressources

Il est utile de se référer à des exemples et à des documentations en ligne pour mieux comprendre et écrire des *Makefiles*.

Exemple

Considérons un projet avec cette structure :



- Le *Makefile* principal dans **`/projet`** pourrait inclure les *Makefiles* de **`module1`** et **`module2`**.
- Chaque sous-répertoire contient un *Makefile* qui compile le code de ce module spécifique.

5. COMMANDES

Les *Makefiles* utilisent une syntaxe particulière pour décrire comment les cibles (généralement des fichiers à construire) peuvent être dérivées des dépendances (généralement des fichiers sources). Voici une introduction aux commandes et directives couramment utilisées dans les *Makefiles* :

5.1. Base

- **`all`**

  Souvent utilisée comme cible par défaut. C'est ce qui est exécuté lorsque vous tapez simplement **`make`**.

Exemple

```
all: mon_programme
```

- **`clean`**

  Une cible courante pour nettoyer les fichiers générés.

Exemple

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

5.2. Compilation

- **`-c`**

Compile les fichiers sources en fichiers objets sans lier.

Exemple

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

- **`-o`**

Spécifie le nom du fichier de sortie.

Exemple

```
mon_programme: main.o
    gcc main.o -o mon_programme
```

5.3. Variables

Les variables sont souvent utilisées pour simplifier les *Makefiles* et les rendre plus modulables.

Exemple

```
# Variables simples
CC = gcc
CFLAGS = -Wall
# Pour les utiliser
main.o: main.c
    $(CC) $(CFLAGS) -c main.c
```

5.4. Directives de contrôle

- **`include`**

Inclut un autre *Makefile*. C'est utile pour séparer les configurations ou partager des règles.

Exemple

```
include mon_second_makefile
```

- **`.PHONY`**

Indique que la cible est fictive et ne représente pas un fichier.

Exemple

```
.PHONY: all clean
```

5.5. Fonctions

Il existe des fonctions intégrées qui peuvent être utilisées pour manipuler des variables ou des listes de fichiers.

- **`wildcard`**

Récupère les noms de fichiers qui correspondent à un motif.

Exemple

```
SOURCES = $(wildcard *.c)
```

- **patsubst**

Remplace un motif par un autre.

Exemple

```
OBJECTS = $(patsubst %.c, %.o, $(SOURCES))
```

5.6. Édition de liens

- **-l**

Lien avec une bibliothèque.

Exemple

```
mon_programme: main.o
    gcc main.o -o mon_programme -lma_librairie
```

- **-L**

Spécifie un répertoire de recherche pour les fichiers de bibliothèque.

Exemple

```
mon_programme: main.o
    gcc main.o -o mon_programme -L/chemin/vers/librairie -lma_librairie
```

- **-I**

Ajoute un répertoire à la liste des répertoires à rechercher pour les en-têtes.

Exemple

```
main.o: main.c
    gcc -I/chemin/vers/en_tetes -c main.c
```

5.7. Autres commandes et options

- **$@**

  Représente le nom de la cible.

Exemple

```
mon_programme: main.o
    gcc main.o -o $@
```

- **$^**

  Représente toutes les dépendances.

Exemple

```
mon_programme: main.o util.o
    gcc $^ -o $@
```

- **$<**

  Représente la première dépendance.

Exemple

```
main.o: main.c
    gcc -c $< -o $@
```

Il s'agit ici d'une introduction de base aux éléments que vous pouvez trouver dans un *Makefile*. Il existe de nombreuses autres fonctionnalités et options avancées que vous pouvez explorer en vous référant à la documentation officielle de *GNU Make*.

6. ÉTAPES DE COMPILATION AVEC *MAKEFILES*

6.1. Compilation

Chaque fichier source (**.c** ou **.cpp**) est compilé en fichier objet (**.o**) à l'aide de la commande de compilation (par exemple, **avr-gcc**).

Exemple

```
module1.o: module1.c
    avr-gcc -c module1.c -o module1.o
```

6.2. Édition de liens

Les fichiers objets sont ensuite regroupés pour former un exécutable.

Exemple

```
main: module1.o module2.o
    avr-gcc module1.o module2.o -o main
```

6.3. Création de librairies statiques

Au lieu de former un exécutable, on peut aussi regrouper les fichiers objets pour créer une librairie statique. Cela est généralement fait avec l'outil **ar**.

Exemple

```
libmodule.a: module1.o module2.o
    ar rcs libmodule.a module1.o module2.o
```

6.4. Utilisation de la librairie

Lors de la création d'un exécutable qui utilise une librairie, celle-ci est spécifiée lors de l'édition de liens.

Exemple

```
main: main.c libmodule.a
    avr-gcc main.c -L. -lmodule -o main
```

7. CONSEILS PRATIQUES

- Utilisez des variables dans vos *Makefiles* pour rendre le code plus propre et modulaire.
- Pour un meilleur suivi, utilisez des commentaires et des affichages informatifs lors de l'exécution de votre *Makefile*.
- Faites preuve de curiosité et explorez d'autres fonctionnalités avancées des *Makefiles* pour améliorer votre efficacité.

Exemple

En utilisant des variables dans le *Makefile* pour une meilleure modularité :

```
CC = avr-gcc
CFLAGS = -I.
OBJFILES = module1.o module2.o
TARGET = main
$(TARGET): $(OBJFILES)
    $(CC) $(OBJFILES) -o $(TARGET)
```

Cet exemple utilise des variables pour le compilateur, les options de compilation, les fichiers objets et la cible, rendant le *Makefile* plus propre et facile à maintenir.

En résumé, les *Makefiles* offrent un moyen puissant d'automatiser et de gérer le processus de compilation et de construction de programmes. En maîtrisant leurs principes et leurs utilisations, vous pouvez grandement améliorer votre efficacité en tant que programmeur, tout en gardant votre code organisé et modulaire.